Mateus Krepsky Ludwich

MÉTODO PARA ABSTRAÇÃO DE COMPONENTES DE HARDWARE PARA SISTEMAS EMBARCADOS

Dissertação submetida ao Programa de Pós Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Antônio Augusto Fröhlich, Prof. Dr.

Florianópolis(SC)

2012

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

L948m Ludwich, Mateus Krepsky
Método para abstração de componentes de hardware para sistemas embarcados [dissertação] / Mateus Krepsky Ludwich ; orientador, Antônio Augusto Fröhlich. - Florianópolis, SC, 2012.
131 p.: il., tabs.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Informática. 2. Ciência da computação. 3. Java (Linguagem de programação de computador). 4. Hardware. I. Fröhlich, Antonio Augusto Medeiros. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 681

MÉTODO PARA ABSTRAÇÃO DE COMPONENTES DE HARDWARE PARA SISTEMAS EMBARCADOS

MATEUS KREPSKY LUDWICH

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Prof. Dr. Ronaldo dos Santos Mello,
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Antônio Augusto Fröhlich,
Orientador

Prof. Dr. Carlos Eduardo Pereira,
Universidade Federal do Rio Grande do Sul

Prof. Dr. Marco Aurélio Wehrmeister,
Universidade Estatual de Santa Catarina

Prof. Dr. Ricardo Pereira e Silva,
Universidade Federal de Santa Catarina

À minha mãe Eleonora Kátia Krepsky Ludwich
e ao meu pai Adriano Brognoli Ludwich

AGRADECIMENTOS

Agradeço ao meu orientador Antônio Augusto Fröhich por acreditar e incentivar o meu trabalho, pelas reuniões de trocas de ideias e pela oportunidade de fazer parte do Laboratório de Integração de Software e Hardware - LISHA.

Agradeço aos amigos do LISHA pelas conversas científicas e pelos momentos de descontração dentro e fora do laboratório.

Aos meus amigos pelas conversas e pela compreensão nos momentos que estive ausente.

Aos meus pais que sempre me apoiaram e continuam me apoiando ao longo da minha vida e carreira acadêmia.

Finalmente agradeço a Deus por proporcionar todas estas coisas e por estar sempre presente em minha vida.

HAL: I'm afraid, Dave.
Dave: I known, HAL, but you've turned into a murderous, psychotic A.I.
Say goodnight...
HAL: GOODDNNIIGGHHT DDAVV...
(Dave shutting HAL down)

Stanley Kubrick - 2001: A Space Odyssey (1968)

RESUMO

Linguagens de programação tem um papel fundamental no desenvolvimento de sistemas computacionais. Dentre elas, as *Linguagens Gerenciadas* (do inglês *Managed Programming Language* - MPL) , dos quais JAVA e LUA são exemplos, possuem funcionalidades que objetivam aumentar a produtividade dos desenvolvedores. Isto é obtido fornecendo construções em um nível mais alto de abstração que permitem expressar e validar ideias em menos tempo e tornando mais difícil a ocorrência de erros de programação o que reduz o tempo gasto na depuração de programas.

Há cerca de uma década diversas iniciativas tem sido executadas com o objetivo de permitir o uso das MPLs não somente em sistemas de propósito geral como também em sistemas embarcados, atendendo requisitos de tempo e consumo de recursos impostos por tais sistemas. Entretanto, para que as MPLs sejam realmente úteis em sistemas embarcados é necessário que elas forneçam funcionalidades para interação com o ambiente no qual o sistema embarcado está inserido. Tal interação usualmente acontece por meio de dispositivos de hardware, como por exemplo, sensores e atuadores, transmissores e receptores, temporizadores e alarmes.

A interação entre MPLs e dispositivos de hardware é realizada por meio das *Interfaces de Função Estrangeira* (do inglês *Foreign Function Interface* - FFI) . Porém, as FFIs em si não especificam como abstrair hardware nem como organizar tais abstrações.

Esta dissertação apresenta um método de como realizar a interface entre dispositivos de hardware e aplicações escritas em MPL para sistemas embarcados. *Mediadores de hardware* são utilizados para abstrair e organizar dispositivos de hardware de forma adequada para sistemas embarcados, cumprindo requisitos de tempo e consumo de recursos. Isolando os mediadores de hardware das especificidades das FFIs o problema de adaptar um dispositivo de hardware para uma nova FFI passa a ser visto como um problema de aplicação de aspectos.

O método proposto é avaliado nas MPLs JAVA e LUA em três casos de estudo, envolvendo aplicações de comunicação serial, codificação de vídeo e monitoramento de temperatura. Os resultados obtidos confirmam a adequação do método nos requisitos de desempenho, consumo de memória, reuso e portabilidade.

Palavras-chave: Sistemas Embarcados, FFI, Java, Lua.

ABSTRACT

Programming Languages have a key role on the development of computational systems. Among them, the so called *Managed Programming Languages* (MPLs), from which JAVA and LUA are examples, provide developers with features to improve their productivity. Productivity improvement is obtained by using constructions with a higher abstraction level, constructions that enable the developer to express and validate his ideas in a short period of time, and by features that make the occurrence of programming errors less often reducing the time spend on program debugging.

Several initiatives have been taken on the last decade in order to enable the use of MPLs not only in general propose systems but also in embedded systems, fulfilling time and resource consumption constraints imposed by these systems. However, in order to be really useful in embedded systems, MPLs must provide features for interacting with the environment in which the embedded system is inserted on. Such interaction is usually implemented by using hardware devices such as, sensors and actuators, transmitters and receivers, and timers and alarms.

The interaction between MPLs and hardware devices is performed by using the so called *Foreign Function Interfaces* (FFIs). However, FFIs by themselves do not specify how to abstract hardware nor how to organize these abstractions.

This dissertation presents a method to interface hardware devices and applications written using MPL in context of embedded systems. *Hardware mediators* are used to abstract and to organize hardware devices in a suitable manner for embedded systems, fulfilling time and resource consumption constraints. By isolating hardware mediators from the specificities of FFIs the problem of adapting a hardware device to work with a new FFI can be faced as a aspect weaving problem.

The proposed method is evaluated on the MPLs JAVA and LUA among three cases study encompassing serial communication, video encoding, and temperature sensing. The obtained results corroborate the suitability of the proposed method on the requirements of performance, memory consumption, reuse, and portability.

Keywords: Embedded Systems, FFI, Java, Lua.

LISTA DE FIGURAS

Figura 1 Decomposição de domínio orientada a aplicação. 32
Figura 2 Família de mediadores de hardware para CPU. 34
Figura 3 Mediadores de hardware dissolvendo-se nas abstrações que os utilizam. 35
Figura 4 Adaptador baseado em classes. 37
Figura 5 Adaptador baseado em objetos. 38
Figura 6 Padrão de projetos *Bridge*. 39
Figura 7 Adaptador de cenário. 41
Figura 8 Elementos estruturais do DERCS. Adaptada de (WEHRMEISTER, 2009). 44
Figura 9 Elementos comportamentais do DERCS. Adaptada de (WEHRMEISTER, 2009). 45
Figura 10 Elementos de orientação a aspecto do DERCS. Adaptada de (WEHRMEISTER, 2009). 46
Figura 11 Passos para a criação de adaptadores de código nativo utilizando a JNI. 54
Figura 12 Classe Java do programa *HelloWorld* utilizando JNI. . . 55
Figura 13 Arquivo de cabeçalho C gerado pela *javah*. 55
Figura 14 Implementação do método *print*. 55
Figura 15 Passos para a criação de adaptadores de código nativo utilizando a KNI. 59
Figura 16 Classe Java do programa Adder utilizando KNI. 60
Figura 17 Implementação do método *sum*. 60
Figura 18 Arquitetura da JVM KESO. Adaptada de (STILKERICH et al., 2006). 62
Figura 19 Processo de geração do sistema. Adaptado de (WAWERSICH; STILKERICH; SCHRÖDER-PREIKSCHAT, 2007). 63
Figura 20 Utilização da FFI do KESO. 65
Figura 21 Arquitetura OSEK/EPOS. 66
Figura 22 Passos para a criação de adaptadores de código nativo utilizando a *NanoVM Native Interface*. 69
Figura 23 Implementação do método *sum* utilizando FFI da NanoVM. 70

Figura 24 Instanciação da LVM e execução de um programa Lua. 74
Figura 25 Parte Lua do programa Adder. 74
Figura 26 Implementação do método *sum* utilizando a API C de Lua. 75
Figura 27 Relação adaptador de código nativo e mediadores de hardware. 82
Figura 28 Adaptador de código nativo antes e depois da aplicação dos aspectos de FFI. 83
Figura 29 MPL e HLL com representação binária distintas. 85
Figura 30 MPL e HLL com a mesma representação binária. 86
Figura 31 Fluxo completo de geração do sistema. 87
Figura 32 Arquitetura do EBG. 89
Figura 33 Extensão da adaptação estrutural do DERCS. 91
Figura 34 Extensão da adaptação comportamental do DERCS. 92
Figura 35 Aplicação de adaptação baseada em objeto. 93
Figura 36 Aplicação de adaptação baseada em classe. 94
Figura 37 Aplicação de adaptação baseada em objeto para a FFI da NanoVM. 95
Figura 38 Aplicação de adaptação baseada em objeto para a FFI da KESO JVM. 96
Figura 39 Classe *weavelet* de um mediador de hardware qualquer. 97
Figura 40 Exemplo UART. 102
Figura 41 Arquitetura do programa UART para a FFI do KESO. 103
Figura 42 Estimativa de movimento. 106
Figura 43 Interação entre *Coordinator* e *Workers*. 107
Figura 44 Adaptador de código nativo para o método *match* (API C de Lua). 108
Figura 45 Adaptador de código nativo para o método *match* (FFI do KESO). 109
Figura 46 Aplicação Java DMEC. 110
Figura 47 Aplicação Lua DMEC. 111
Figura 48 Aplicação de monitoração de temperatura. 112
Figura 49 Aplicação *sensor*. 113
Figura 50 Aplicação *sink*. 114

LISTA DE TABELAS

Tabela 1 *Overhead* de tempo gerado pelo adaptador de código nativo. 104
Tabela 2 *Overhead* de memória para *UART::put*. 105
Tabela 3 *Overhead* de tempo gerado pelo adaptador de código nativo de *DMEC::match*. 110
Tabela 4 *Overhead* de memória para *DMEC::match*. 111
Tabela 5 *Overhead* de tempo gerado pelos adaptadores de código nativo. Arquitetura AVR8, FFI da KESO. 115
Tabela 6 *Overhead* de tempo gerado pelos adaptadores de código nativo. Arquitetura ARM7, FFI da NanoVM. 115
Tabela 7 *Overhead* de memória. Arquitetura AVR8, FFI da KESO 116
Tabela 8 *Overhead* de memória. Arquitetura ARM7, FFI da NanoVM 117
Tabela 9 Síntese das avaliações realizadas. 118

LISTA DE SIGLAS

ADESD *Application-Driven Embedded System Design*

AMoDE-RT *Aspect-oriented Model-Driven Engineering for Real-Time systems*

AOP *Aspect-Oriented Programming*

API *Application Programming Interface*

AST *Abstract Sintax Tree*

CBA *Class-based Adaptation*

CLDC *Connected Limited Device Configuration*

CPU *Central Processing Unit*

CRC *Cyclic Redundancy Check*

DERCS *Distributed Embedded Real-time Compact Specification*

DMEC *Distributed Motion Estimation Component*

EBG *Extensible Binding Generator*

EPOS *Embedded Parallel Operating System*

FFI *Foreign Function Interface*

HAL *Hardware Abstraction Layer*

HLL *High-Level Language*

IP *Intellectual Property*

JIT *Just-In-Time*

JME *Java Micro Edition*

JNI *Java Native Interface*

JSE *Java Standard Edition*

JVM *Java Virtual Machine*

KCL *KESO Configuration Language*

KNI *K Native Interface*

KVM *K Virtual Machine*

LVM *Lua Virtual Machine*

MDE *Model-Driven Engineering*

ME *Motion Estimation*

MMU *Memory Management Unit*

MPU *Memory Protection Unit*

MPL *Managed Programming Language*

MPSoC *Multiprocessor System-on-Chip*

NIC *Network Interface Card*

NPL *Native Programming Language*

OBA *Object-based Adaptation*

OIL *OSEK Implementation Language*

RSSF Rede de Sensores em Fio

SBTVD Sistema Brasileiro de TV Digital

SE Sistema Embarcado

SO Sistema Operacional

SWIG Simplified Wrapper and Interface Generator

UART *Universal Asynchronous Receiver Transmitter*

UML *Unified Modeling Language*

VM *Virtual Machine*

SUMÁRIO

1 INTRODUÇÃO 23
1.1 OBJETIVOS 25
1.2 METODOLOGIA 26
1.3 ESTRUTURA DA DISSERTAÇÃO 26
2 ABSTRAÇÃO DE DISPOSITIVOS DE HARDWARE 29
2.1 CAMADA DE ABSTRAÇÃO DE HARDWARE 29
2.2 MEDIADORES DE HARDWARE E ADESD 31
2.3 TÉCNICAS DE ADAPTAÇÃO DE INTERFACE 36
2.3.1 Padrões de Projeto para Adaptação de Interface . . . 36
2.3.2 Adaptadores de Cenário 38
2.3.3 DERCS 42
2.4 LINGUAGEM DE PROGRAMAÇÃO GERENCIADA 47
3 AMBIENTES DE SUPORTE À EXECUÇÃO EM MPLS 51
3.1 INTERFACES DE FUNÇÃO ESTRANGEIRA PARA JAVA E LUA 51
3.1.1 Java Native Interface 52
3.1.1.1 HotSpot VM 52
3.1.1.2 JNI 53
3.1.2 K Native Interface 56
3.1.2.1 KVM 56
3.1.2.2 KNI 57
3.1.3 KESO Native Interface 61
3.1.3.1 JVM KESO 61
3.1.3.2 FFI da KESO 64
3.1.3.3 Integração com o EPOS 65
3.1.4 NanoVM Native Interface 66
3.1.4.1 NanoVM 67
3.1.4.2 FFI da NanoVM 67
3.1.4.3 Integração com o EPOS 71
3.1.5 Lua Foreing Function Interface 71
3.1.5.1 LVM 71
3.1.5.2 API C 73
3.1.5.3 Integração com o EPOS 75
3.2 GERAÇÃO AUTOMÁTICA DE ADAPTADORES DE CÓDIGO NATIVO 76
3.3 DEPURAÇÃO DE ADAPTADORES DE CÓDIGO NATIVO 77
3.4 DISCUSSÃO 77

4 ABSTRAÇÃO DE COMPONENTES DE HARDWARE PARA MPLS EMBARCADAS 79
4.1 REQUISITOS 79
4.2 CONCEPÇÃO DO MÉTODO 80
4.2.1 Portabilidade 80
4.2.2 Reuso entre FFIs Distintas 81
4.2.3 Desempenho e Consumo Eficiente de Recursos 83
4.3 PROJETO E IMPLEMENTAÇÃO DO MÉTODO 85
4.3.1 Representação de Mediadores e de Aspectos de FFI 90
4.3.2 Composição de Mediadores e Aspectos de FFI 91
4.3.3 Geração Automática de Adaptadores de Código Nativo 94
5 AVALIAÇÃO DO MÉTODO PROPOSTO 99
5.1 DEFINIÇÃO DAS MÉTRICAS UTILIZADAS 99
5.1.1 Desempenho 99
5.1.2 Consumo de Memória 100
5.1.3 Portabilidade 100
5.1.4 Reuso entre FFIs Distintas 101
5.2 CASOS DE ESTUDO 101
5.2.1 Aplicação Comunicação Serial 101
5.2.2 Aplicação Estimativa de Movimento Distribuída ... 105
5.2.3 Aplicação de Monitoração de Temperatura 112
5.3 DISCUSSÃO 118
6 CONCLUSÕES 121
REFERÊNCIAS 125

1 INTRODUÇÃO

Linguagem de Programação Gerenciada (do inglês *Managed Programming Language* - MPL) , é a denominação dada a linguagens de programação que apresentam um conjunto de funcionalidades além das linguagens de alto nível convencionais, visando aumentar a produtividade do desenvolvedor que as utiliza e reduzindo o tempo das atividades de codificação e depuração. Entre tais funcionalidades estão o gerenciamento automático e proteção de memória, e a tipagem forte. Por meio de proteção de memória, o desenvolvedor pode ser avisado que está tentando acessar uma posição inválida de um vetor ou um objeto não iniciado. O uso de coletores de lixo libera o desenvolvedor da tarefa de escrever código para desalocar objetos e elimina a possibilidade de vazamento de memória. A tipagem forte promove a *segurança de tipo*, impedindo operações entre variáveis de tipos distintos que possam acarretar em resultados inesperados e erros difíceis de detectar (BOND; MCKINLEY, 2009; PIZLO et al., 2010; ESMAEILZADEH et al., 2011; PHIPPS, 1999).

É possível estender o uso de MPLs para o desenvolvimento não somente de aplicações computacionais de propósito geral, como também para o desenvolvimento de sistemas computacionais embarcados. Entretanto, para que isso seja viável é necessário que as implementações das MPLs respeitem os requisitos impostos pelo Sistema Embarcado (SE) alvo. Dentre estes requisitos estão desempenho, consumo de memória, consumo de energia e requisitos de tempo real.

Diversas são as abordagens utilizadas para tratar cada requisito de SE. Para aumento de desempenho e redução do consumo de memória dos ambientes de suporte a execução das MPLs, pode-se empregar o uso de máquinas virtuais dedicadas (BROUWERS; LANGENDOEN; CORKE, 2009; HARBAUM, 2005), técnicas de compilação e geração de sistema (THOMM et al., 2010; FRENZ, 2011; PIZLO et al., 2010), ou ainda o uso de hardware dedicado (ARM, 2011; SCHOEBERL, 2008; PUFFITSCH; SCHOEBERL, 2007). Utilizando-se modelos de execução previsíveis, análises de escalonabilidade podem ser executas e requisitos de tempo real garantidos (BØGHOLM et al., 2010; ZERZELIDIS; WELLINGS, 2010; BØGHOLM et al., 2009). Explorando as configurações de hierarquia de memória e os algoritmos de gerenciamento de memória é possível estimar consumo de energia (SAMPSON et al., 2011; VELASCO; ATIENZA; OLCOZ, 2009).

Para que uma linguagem de programação seja útil no desenvol-

vimento de sistemas embarcados, além de ser adequada aos requisitos impostos por estes sistemas, é necessário que ela forneça ao desenvolvedor funcionalidades que permitam a interação com o ambiente no qual o sistema embarcado está inserido. Esta interação é realizada em sua base por dispositivos de hardware. *Sensores* e *atuadores* são utilizados para interagir com o ambiente no qual o SE está inserido. *Transmissores* e *receptores* são utilizados como base para a comunicação entre SEs. *Temporizadores* são utilizados para implementar operações de tempo real.

Dispositivos de hardware são acessados por meio de leituras e escritas em registradores específicos mapeados em memória ou pelo uso de instruções dedicadas de entrada e saída. Em linguagens de alto nível como C/C++ isso é resolvido, respectivamente, com a utilização de ponteiros e com *inlining* de instruções *assembly*. Entretanto, como as MPLs possuem gerência automática de memória, construções como ponteiros não são fornecidas. A solução adotada pelas MPLs é chamada de Interface de Função Estrangeira (do inglês *Foreign Function Interface* - FFI) , por meio da qual é possível acessar construções de outras linguagens de programação (como os ponteiros de C/C++) para acessar dispositivos de hardware, quando necessário. O código escrito utilizando-se uma FFI, o qual concentra o conhecimento de acesso a um determinado dispositivo de hardware, é chamado de adaptador de código nativo (*binding code*).

A utilização das FFIs por si só não auxilia na abstração dos dispositivos de hardware que se deseja acessar. Por falta de um padrão de como realizar esta abstração, a interface entre dispositivos de hardware e MPLs é realizada de forma *ad hoc*. Como detalhes do dispositivo de hardware estão presentes no adaptador de código nativo, a portabilidade do adaptador para diferentes plataformas de hardware torna-se difícil. Além disso, o adaptador de código nativo desenvolvido utilizando-se uma determinada FFI dificilmente pode ser utilizado em outra FFI sem significativas alterações uma vez que este usa métodos específicos da API da FFI na sua construção.

No cenário de computação de propósito geral o problema de portabilidade de adaptadores de código nativo pode ser resolvido utilizando-se de uma camada de abstração de hardware (do inglês *Hardware Abstraction Layer* - HAL) . Uma HAL encapsula todos os recursos disponíveis em uma plataforma, o que gera interdependências entre os dispositivos de hardware abstraídos. O *overhead* decorrente desta interdependência, frequentemente inviabiliza a utilização de HALs em um cenário de computação embarcada.

A metodologia de Projeto de Sistemas Embarcados Orientado pela Aplicação (do inglês *Application-Driven Embedded System Design* - ADESD) (FRöHLICH, 2001) apresenta o conceito de *mediadores de hardware*. Mediadores de hardware sustentam um *contrato de interface* entre as abstrações de sistema e a plataforma de hardware, permitindo às abstrações independência de plataforma. Existe um medidor de hardware para cada dispositivo de hardware sendo abstraído, evitando-se assim o *overhead* ocasionado pelas HALs tradicionais. Aliado a isso, a utilização de técnicas de meta programação e *inlining* de funções na implementação dos mediadores permitem dissolve-los entre as abstrações que os utilizam, reduzindo o *overhead* de tempo no uso de mediadores e tornando-os bons candidatos para serem utilizados em sistemas embarcados (POLPETA; FRöHLICH, 2004).

1.1 OBJETIVOS

A hipótese de pesquisa que norteia esta dissertação é que a utilização do conceito de mediadores de hardware, proposto pela ADESD, em conjunto com uma FFI focada em SEs constitui um método adequado para prover dispositivos de hardware para aplicações embarcadas escritas utilizando-se MPLs. Desta forma, o principal objetivo deste trabalho é demonstrar a veracidade desta hipótese de pesquisa. Não é objetivo deste trabalho propor uma nova implementação de MPL para sistemas embarcados e sim propor e demostrar a eficiência de um método capaz de ser utilizado por várias FFIs e várias MPLs distintas para abstrair o acesso a dispositivos de hardware no domínio de sistemas embarcados.

Foram identificados os seguintes objetivos específicos, os quais suportam o objetivo principal deste trabalho:

- Demostrar a utilização de mediadores de hardware como forma de encapsular especificidades de plataformas de hardware, promovendo portabilidade de um adaptador de código nativo entre diferentes plataformas.

- Demostrar que o uso de mediadores de hardware permite criar um novo nível de abstração no desenvolvimento de adaptadores de código nativo sem acrescentar *overhead* de tempo e de memória significativos.

- Demostrar que, por meio de técnicas de programação orientada a aspectos, é possível fatorar especificidades de FFIs distintas em

programas de aspectos e isolá-las da parte funcional dos adaptadores de código nativo promovendo o reuso dos mesmos entre diferentes FFIs.

Assim, a principal contribuição deste trabalho é um método de como realizar a interface entre dispositivos de hardware e MPLs no contexto de sistemas embarcados. Este método pode ser utilizado para construir bibliotecas de componentes de hardware para serem utilizados em aplicações escritas em MPLs para SEs.

1.2 METODOLOGIA

A metodologia utilizada para verificar se os objetivos propostos neste trabalho foram atingidos é a realização de experimentos comparando a abordagem proposta com trabalhos existentes.

As MPLs escolhidas para a realização dos experimentos foram a linguagens JAVA e LUA. Para o propósito deste trabalho consideram-se como MPLs todas as linguagens de alto nível que possuam gerência automática de memória, encapsulando os acessos aos endereços de memória dentro do seu ambiente de execução.

Os mediadores de hardware utilizados nos experimentos estão implementados no Sistema Operacional Paralelo e Embarcado (do inglês *Embedded Parallel Operating System* - EPOS) , que representa o caso estudo da aplicação da ADESD no domínio de sistemas operacionais.

A metodologia utilizada neste trabalho pode ser dividida nas seguintes etapas:

- Exportar para as MPLs alvo um conjunto de mediadores de hardware utilizando FFIs focadas em SEs.

- Fatorar em aspectos as especificidades das FFIs, isolando-as da parte funcional dos adaptadores de código nativo.

- Comparar aplicações que acessam dispositivos de hardware utilizando a proposta deste trabalho, com outros trabalhos já existentes. Nestas comparações serão avaliados *overhead* de tempo de invocação de métodos, *overhead* de memória e portabilidade.

1.3 ESTRUTURA DA DISSERTAÇÃO

Os capítulos que seguem estão organizados nesta dissertação da seguinte forma.

O Capítulo 2 corresponde à fundamentação teórica do trabalho. Neste capítulo é apresentado como dispositivos de hardware são abstraídos e organizados, são revisadas as principais técnicas de adaptação de interface entre componentes e é apresentado o conceito de Linguagem de Programação Gerenciada e como tais linguagens interagem com dispositivos de hardware utilizando o conceito de Interface de Função Estrangeira.

O Capítulo 3 revisa o estado da arte, apresentando as principais FFIs de JAVA e de LUA e apresentando geradores de adaptadores de código nativo e como os adaptadores de código nativo podem ser verificados quanto a sua correção.

O Capítulo 4 apresenta a proposta desta dissertação de como realizar a interface entre dispositivos de hardware e aplicações escritas em MPLs no contexto de SEs.

O Capítulo 5 apresenta os experimentos realizados, define métricas de desempenho, consumo de memória, portabilidade e reúso de adaptadores de código nativo entre FFIs distintas, as quais são utilizadas na comparação com trabalhos relacionados e discute os resultados obtidos nessas comparações.

O Capítulo 6 finaliza a dissertação apresentando as conclusões e apontando os trabalhos futuros.

2 ABSTRAÇÃO DE DISPOSITIVOS DE HARDWARE

Dispositivos de hardware são abstraídos para serem utilizados por software basicamente de duas maneiras: por mapeamento em memória e pelo uso de instruções específicas. Um programa pode controlar um dispositivo de hardware mapeado em memória lendo e escrevendo em endereços de memória específicos para os quais o dispositivo foi configurado. É possível também controlar dispositivos de hardware por meio de instruções específicas, como por exemplo, instruções de entrada e saída, que podem ser utilizadas para ler e escrever em um dado dispositivo. Os programas que tem a funcionalidade específica de controlar dispositivos de hardware são chamados de controladores de dispositivos (do inglês *device drivers*).

Neste capítulo apresentam-se as abordagens pelas quais os dispositivos de hardware podem ser abstraídos e organizados de forma que eles possam ser utilizados diretamente pela aplicação ou por abstrações de software de mais alto nível. No cenário de sistemas embarcados a proximidade das aplicações com os controladores de dispositivos como alternativa a mecanismos mais elaborados como as chamadas de sistemas UNIX é justificada não somente pelas restrições de recursos computacionais mas, principalmente porque tais sistemas executam apenas uma única aplicação específica e dedicada. Nesse sentido, são apresentadas as camadas de abstração de hardware, que constituem uma forma tradicional para abstrair e agrupar os recursos de hardware de uma plataforma, mas em geral não utilizam de metodologias de engenharia de software para sua concepção e não são adequadas para SEs. Como alternativa às camadas de abstração de hardware tradicional, discute-se brevemente a metodologia de Projeto de Sistemas Embarcados Orientado pela Aplicação e apresenta-se o conceito de mediadores de hardware. Neste capítulo apresentam-se também quais são as técnicas utilizadas para adaptar interfaces de componentes. Por fim, é apresentado o conceito de Linguagem de Programação Gerenciada e como estas linguagens interagem com dispositivos de hardware.

2.1 CAMADA DE ABSTRAÇÃO DE HARDWARE

Camada de Abstração de Hardware (do inglês *Hardware Abstraction Layer* - HAL) é uma das formas para abstrair especificidades de hardware, apresentando ao sistema operacional, dispositivos abs-

tratos de hardware. A abstração dos dispositivos de hardware se dá por meio de serviços como tratamento de interrupções, tratamento da reinicialização, transferências DMA, controle de temporizadores e sincronismo entre multiprocessadores (TANENBAUM, 2007).

É possível, por meio do uso de HALs escrever *drivers* que utilizem operações abstratas de entrada e saída, as quais são ou mapeadas em operações reais de entrada e saída de dispositivos mapeados em memória ou mapeadas em operações utilizando-se portas especiais de entrada e saída, de acordo com a disponibilidade da plataforma de hardware.

O porte da HAL em si para uma nova plataforma de hardware significa implementar todos os serviços que a HAL disponibiliza. Entretanto, muitos dos serviços disponíveis por uma HAL podem não ser utilizados por uma aplicação e fornecer todos estes serviços pode gerar um *overhead* de memória e desempenho desnecessário, conforme demostrado por (POLPETA, 2006), que identifica três exemplos de *overhead* decorrentes pela forma em que as HALs são projetadas e implementadas.

O primeiro exemplo ocorre no gerente de memória em sistemas UNIX. A chamada de sistema *brk*, utilizada para redimensionar a área de dados utilizada por um processo, sempre pressupõe que o modelo paginado de memória está em uso e, consequentemente, de que a plataforma disponha de uma Unidade de Gerenciamento de Memória (do inglês *Memory Management Unit* - MMU) . Isto compromete a portabilidade destes sistemas para plataformas que não disponham de MMU.

O segundo exemplo ocorre no uCLinux, um sistema operacional voltado para sistemas embarcados que se fundamenta em uma HAL. Neste caso, herda-se do Linux diversas das suas funcionalidades, dentre elas a abstração de sistema de arquivos. Isso impacta não somente no tamanho do sistema operacional gerado, como também em toda a infraestrutura de inicialização do sistema e carga das aplicações.

O terceiro exemplo ocorre no sistema operacional eCos da empresa RedHat. A HAL utilizada no eCos é baseada em componentes de software, entretanto ela não é gerada de acordo com a aplicação, podendo agregar código desnecessário ao sistema. No caso de SoCs gerados a partir da microarquitetura LEON2, por exemplo, o sistema eCos tem como certa a existência de uma UART, o que não necessariamente é verdadeiro.

A problemática do uso de HALs é ainda maior em cenários onde a arquitetura de hardware pode ser modificada em um curto espaço de tempo. Este é o caso quando o hardware a ser abstraído é instanciado em dispositivos de lógica programável, como FPGAs (POLPETA, 2006).

Como alternativa ao desenvolvimento tradicional de HALs, metodologias que promovam engenharia de domínio para o desenvolvimento de componentes de software têm sido utilizadas para abstrair dispositivos de hardware e organizar tais abstrações sem gerar interdependências desnecessárias entre tais dispositivos. Este é o caso da Metodologia de Projeto de Sistemas Embarcados Orientado pela Aplicação, apresentada na Seção 2.2.

2.2 MEDIADORES DE HARDWARE E ADESD

O conceito de mediadores de hardware é um dos conceitos definidos pela metodologia de Projeto de Sistemas Embarcados Orientado pela Aplicação (do inglês *Application-Driven Embedded System Design* - ADESD) (FRöHLICH, 2001). Portanto, antes de apresentar a definição de mediador de hardware apresenta-se brevemente a ADESD focando nos conceitos utilizados neste trabalho.

A ADESD guia a concepção e o desenvolvimento de sistemas em embarcados dedicados, o quais executarão uma única aplicação. Apesar do sistema embarcado alvo executar uma única aplicação, a ADESD não é uma metodologia centrada no desenvolvimento de uma aplicação por vez, pelo contrário, utilizando de análise de domínio a ADESD propõe o desenvolvimento de *frameworks* os quais permitirão o desenvolvimento de potencialmente todas as aplicações pertencentes a um mesmo domínio.

O passo inicial da ADESD é a identificação das entidades presentes no domínio as quais são modeladas como abstrações. A Figura 1 mostra como o domínio é decomposto em abstrações. As abstrações são agrupadas em famílias de acordo com as similaridades entre elas. As funcionalidades em comum dos membros de uma família são agrupadas na chamada *interface inflada*, definindo um “macro componente” que representa a família. Utilizando interfaces infladas os desenvolvedores podem projetar a aplicação livres da preocupação de qual membro da família irá ser utilizado em cada momento.

Uma abstração, identificada durante a análise de domínio, pode ser utilizada em diferentes cenários de execução. Se no projeto das abstrações identificadas durante a análise for considerado apenas um cenário específico, estas abstrações serão dificilmente portadas para outros cenários. A ADESD propõe a solução deste problema fatorando os cenários em que uma abstração pode executar, em aspectos. Por exemplo, um componente de comunicação (abstração), é projetado para ser

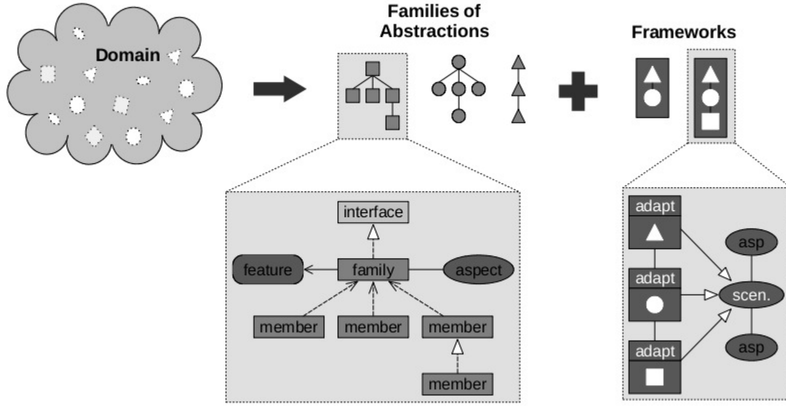

Figura 1: Decomposição de domínio orientada a aplicação.

independente de cenário. Um cenário para este componente seria o cenário de execução *multithreading*. Desta forma, quando habilitado o aspecto *multithreading*, o componente de comunicação passaria, por exemplo, e ficar bloqueado na execução de métodos que envolvam uma região crítica. A composição (*weaving*) dos aspectos com as abstrações geraram novos membros de uma família de componentes, eliminando o custo de implementar um novo membro para cada abstração de uma família quando um novo cenário fosse identificado, o que poderia levar a uma explosão combinatória no número de membros de uma família.

Os aspectos e as abstrações são compostos utilizando-se adaptadores de cenário, os quais serão descritos na Seção 2.3.2. Um conjunto de adaptadores de cenário e aspectos de cenário forma um *framework* que específica como as abstrações são interconectadas.

No contexto de um determinado cenário de aplicação, componentes podem ter alguns de seus parâmetros modificados, habilitando ou desabilitando funcionalidades do componente. Isto é obtido através do conceito de *propriedades configuráveis* (*configurable features*). Um exemplo de propriedades configurável é o tamanho de segmento de pilha das *threads* que comporão uma aplicação.

O Sistema Operacional Paralelo e Embarcado (do inglês *Embedded Parallel Operating System* - EPOS) representa o caso de estudo da aplicação da ADESD no domínio de sistemas operacionais (FRöHLICH, 2001). No caso do EPOS as abstrações decorrentes da decomposição de domínio são abstrações típicas de Sistema Operacional (SO). São exem-

plos de abstrações: *Threads*, que representam fluxos de execução em um programa; abstrações de sincronização como semáforos, variáveis de condição e *mutexes*; e abstrações de comunicação como redes, canais, etc.

Mediadores de Hardware

Para fornecer o suporte de hardware necessário para a implementação das abstrações, sem que estas venham a ser dependentes de hardware, a ADESD define o conceito de *mediadores de hardware*. Os mediadores de hardware são semelhantes aos *device drivers* UNIX, no sentido de que eles abstraem dispositivos de hardware para serem utilizados por outros componentes do sistema.

Mediadores de hardware sustentam um *contrato de interface* entre as abstrações de sistema e a plataforma de hardware, permitindo às abstrações independência de plataforma (POLPETA; FRöHLICH, 2004). Este contrato de interface além de ocultar especificidades de dispositivos de hardware de fabricantes ou plataformas distintos, pode prover em software funcionalidades que não estejam originalmente presentes no dispositivo de hardware utilizado. Por exemplo, um mediador de Interface de Rede (do inglês *Network Interface Card* - NIC) pode prover um método para geração de CRC, um código utilizado para verificar a integridade na transmissão de quadros em uma rede *ethernet*, independentemente do dispositivo de hardware em questão possuir ou não tal funcionalidade. Ou seja, a abstração que utilizará o mediador da NIC poderá sempre assumir que o método para calculo de CRC existe e ficará a cargo da implementação do mediador delegar este calculo para hardware, caso exista esta funcionalidade no dispositivo de hardware em questão ou, caso contrário, de implementar esta funcionalidade em software.

Assim como as abstrações de sistema, os mediadores de hardware podem ser organizados em famílias. A Figura 2 mostra a família dos mediadores de hardware de uma Unidade Central de Processamento (do inglês *Central Processing Unit* - CPU) . Uma interface inflada agrupa os métodos que estarão presentes em todos os membros da família CPU. Estes métodos realizam carga e salvamento de contexto, desativam e ativam interrupções e realizam operações atômicas de incremento e decremento. Utilizando a interface inflada da família CPU, a abstração *Thread*, pode ser desenvolvida abstraindo-se dos detalhes da CPU real na qual ela executará.

Existe um mediador de hardware para cada dispositivo de hardware presente na plataforma. Este nível fino de granularidade representa a principal diferença entre os mediadores de hardware e HALs

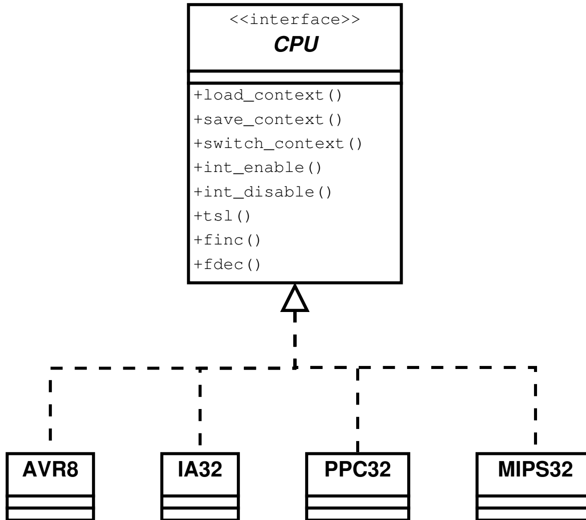

Figura 2: Família de mediadores de hardware para CPU.

tradicionais, as quais são desenvolvidas de maneira monolítica gerando falsas interdependências entre dispositivos de hardware, como os exemplos mencionados na Seção 2.1.

A Figura 3 mostra a granularidade dos mediadores, um para cada dispositivo de hardware. A figura ilustra também outro aspecto importante no desenvolvimento dos mediadores: a ausência de *overhead* de chamada aos seus métodos. Isso é obtido utilizando-se de técnicas de metaprogramação estática baseada em *templates*, juntamente com a funcionalidade de *inlining* de funções provida na linguagem C++, as quais fazem com que os mediadores se dissolvam nas abstrações que os utilizam.

De forma similar às abstrações de sistemas, aspectos também podem ser aplicados sobre famílias de mediadores de hardware. Por exemplo, as famílias UART e NIC devem frequentemente operar em modo exclusivo de acesso; isto poderia ser alcançado por meio da aplicação de um aspecto de *controle de acesso* sobre estas famílias. A proposta apresentada no Capítulo 4 deste trabalho, fatora especificidades de diferentes FFIs em aspectos, os quais são aplicados sobre mediadores de hardware, adaptando-os à uma MPL específica.

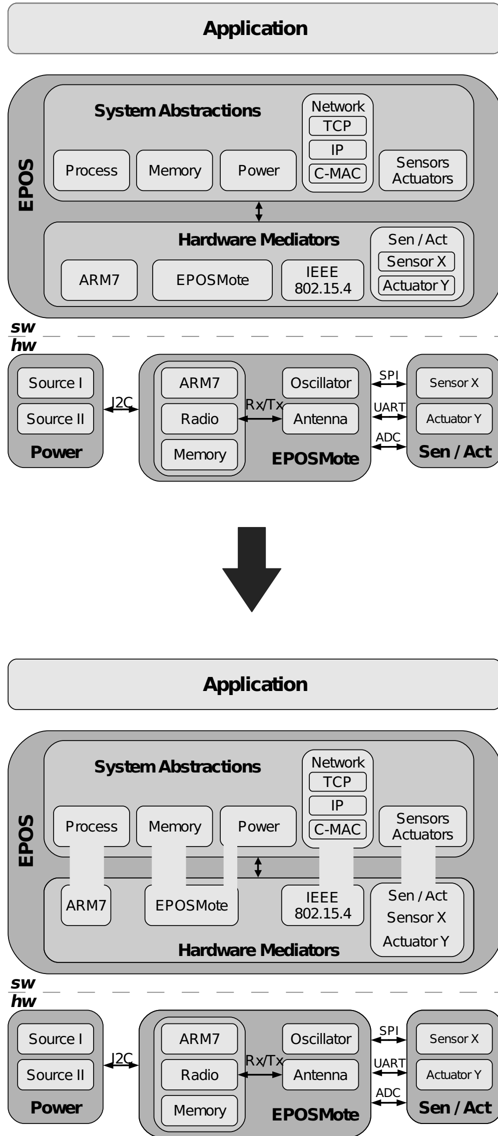

Figura 3: Mediadores de hardware dissolvendo-se nas abstrações que os utilizam.

2.3 TÉCNICAS DE ADAPTAÇÃO DE INTERFACE

Esta seção apresenta técnicas utilizadas para adaptar interfaces de componentes. Tais técnicas serviram de inspiração para o método de abstração de componentes de hardware para MPLs distintas, apresentado no Capítulo 4. Inicialmente são revisados padrões de projetos relacionados com a adaptação de interface. Em seguida, discutem-se como conceitos de AOP podem ser utilizados para adaptar interfaces e apresenta-se o padrão *adaptador de cenário* proposto pela ADESD. Por fim, discute-se o metamodelo DERCS e como este pode ser utilizado para adaptar interfaces de componentes.

2.3.1 Padrões de Projeto para Adaptação de Interface

Diversos padrões orientados a objetos podem ser utilizados na adaptação de interfaces de componentes, sendo o mais evidente deles o padrão Adaptador (*Adapter*) (GAMMA et al., 1995). Suponha que uma aplicação deseje utilizar um componente previamente disponível, mas não pode fazê-lo, pois a interface deste componente é diferente da interface que ela espera. Uma possível solução seria alterar a interface do componente para deixa-la como a aplicação espera. Contudo, isto só é possível de ser feito se o código fonte do componente estiver disponível. Além disso, alterar um componente previamente disponível cuja interface seja bem aceita pelos utilizadores do componente apenas para satisfazer uma aplicação específica pode não fazer sentido. O padrão Adaptador serve para resolver esta incompatibilidade de interfaces sem alterar o componente original.

Existem duas versões do padrão de projeto Adaptador, a versão baseada em classe e a versão baseada em objeto, apresentadas, respectivamente, nas Figuras 4 e 5. Ambas as versões do padrão possuem a mesma finalidade, diferindo apenas em estrutura. A aplicação, nas Figuras 4 e 5 é representada pela classe *Client* e necessita utilizar um objeto que possua a interface especificada por *Target*. A versão baseada em classes do Adaptador utiliza de herança para compor as classes do adaptador (*Adapter*) e do objeto que representa o componente previamente disponível, o qual será adaptado (*Adaptee*). Na versão baseada em objeto do Adaptador, ao invés de herança é utilizada delegação para compor as classes *Adapter* e *Adaptee*. A versão baseada em classes do Adaptador tem a vantagem de ser, em princípio, mais eficiente que a versão baseada em objetos, pois ela não contém a indireção causada

pelo acesso ao objeto *adaptee*, como mostra a Figura 5. Por outro lado, a versão baseada em objetos do adaptador tente a ser mais flexível, pois é possível, em tempo de execução, trocar o componente a ser adaptado.

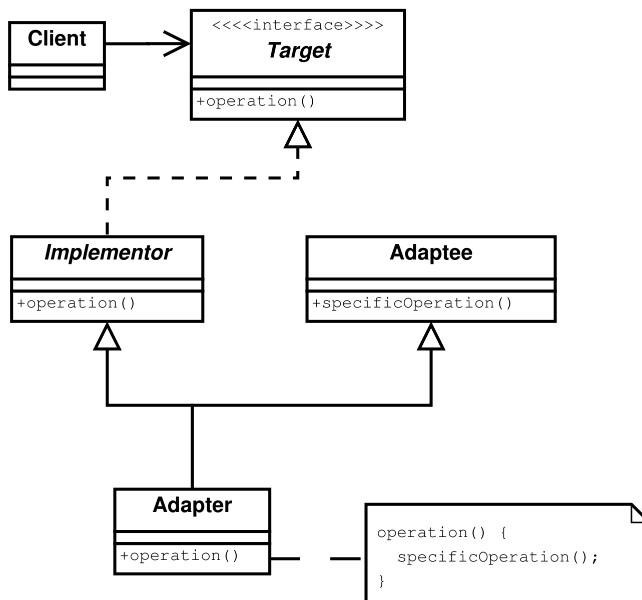

Figura 4: Adaptador baseado em classes.

Outro padrão de projeto que pode ser utilizado na adaptação de interfaces de componentes é o padrão "Ponte" (*Bridge*), apresentado na Figura 6 (GAMMA et al., 1995). O objetivo do padrão *Bridge* é o de separar interface de implementação e permitir que ambas evoluam separadamente. Na Figura 6 a classe abstrata *Abstraction* possui uma referencia a um objeto do tipo *Implementor*, o qual é utilizado para implementar todas as operações de *Abstraction*. Desta forma, as classes *Abstraction* e *Implementor* podem ser refinadas de forma independente uma da outra.

As subclasses de *Abstraction* fornecem as operações de acordo com a interface esperada pelo objeto da classe *Client*, que representa a aplicação. Neste sentido, tais subclasses adaptam a interface dos implementadores à interface que a aplicação necessita.

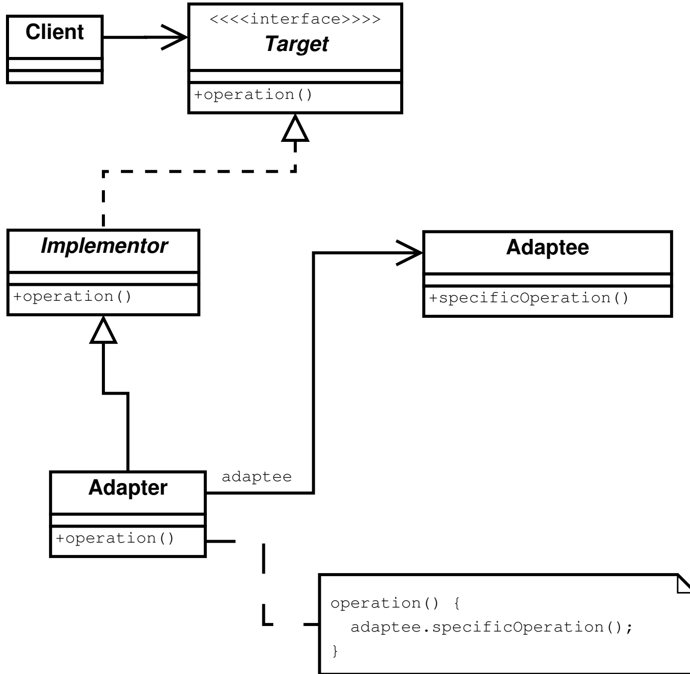

Figura 5: Adaptador baseado em objetos.

2.3.2 Adaptadores de Cenário

A questão da separação de responsabilidades (*separation of concerns*) no desenvolvimento de sistemas computacionais é antiga (DIJKSTRA, 1982). Ela visa o desenvolvimento de abstrações de forma em que cada aspecto envolvendo a abstração possa ser desenvolvido de forma independente. A principal vantagem do desenvolvimento de abstrações independentes do cenário em que elas serão utilizadas é o reuso das mesmas em diferentes cenários de aplicação. Além do reuso outra vantagem da separação de responsabilidades é facilitação da manutenção e evolução das abstrações.

É necessário um suporte por parte da engenharia de software para que a separação entre as abstrações e os cenários nos quais elas executarão seja mantido durante todas as fases de desenvolvimento. Um exemplo é observado por (FRöHLICH; Schröder-Preikschat, 2000) no domínio de sistemas operacionais. A partir da fase de análise obtém-

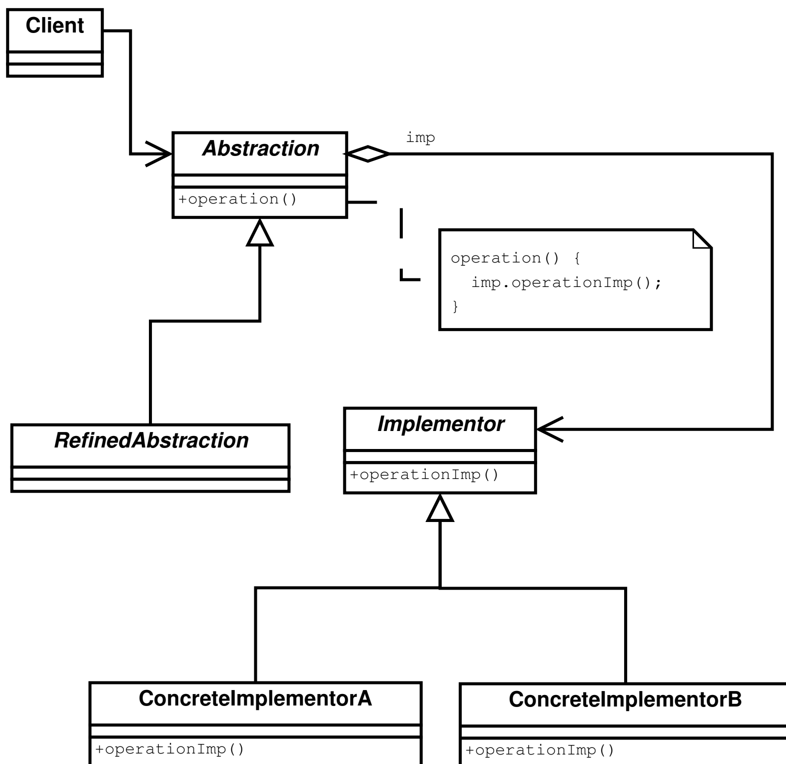

Figura 6: Padrão de projetos *Bridge*.

se que uma caixa de mensagens (*mailbox*) é uma abstração que provê comunicação bidirecional N para N entre objetos ativos. Na fase de projeto é natural que a especificação da caixa de mensagens, proveniente da fase de análise, seja estendida para incluir métodos de envio e recepção de mensagens, construtores que permitirão identificação da caixa de mensagens, armazenamento de mensagens, entre outros. Porém, se não forem tomadas as devidas precauções, na fase de implementação a abstração de caixa de mensagens pode ser poluída com detalhes de possíveis camadas subjacentes, com questões de gerenciamento de *buffers*, segurança, e muitas outras, transformando a abstração caixa de mensagem em uma abstração dependente de cenário.

A Programação Orientada a Aspectos (do inglês *Aspect-Oriented Programming* - AOP) provê conceitos para suportar a questão de separação de responsabilidades (KICZALES et al., 1997). A AOP identifica

responsabilidades que são transversais (*cross-cuting concerns*) a várias abstrações de um sistema. Tais responsabilidades geralmente estão associadas a requisitos não funcionais de um sistema, como segurança, persistência, tratamento de exceções, registro (*logging*), entre outros. A AOP propõe o encapsulamento destas responsabilidades transversais em unidades chamadas de *aspectos*.

Na terminologia da AOP, os pontos de um programa que serão afetados pelas responsabilidades transversais são chamados de pontos de junção (*join points*). Um ponto de junção é especificado e quantizado por meio de um ponto de corte (*point cut*). O chamado conselho (*advice*) representa o código que será inserido nos pontos de junção, afetando o programa em termos de comportamento ou estrutura. A combinação dos pontos de junção com os conselhos é chamada de aspecto. O processo de composição de aspectos é chamado de *aspect weaving*.

Embora a AOP forneça o suporte necessário para realizar separação de responsabilidades, a AOP por si só não especifica como desenvolver abstrações independentes de cenário. A decomposição de domínio orientada a aplicação proposta pela ADESD, se apresenta como uma solução para esse questão, demonstrando como extrair abstrações independentes de cenário de um domínio e como compô-las com os possíveis cenários de aplicação.

O adaptador de cenário é um padrão de projeto proposto pela ADESD que especifica como pode ser realizada a composição de abstrações independentes de cenário com os cenários onde estas abstrações serão aplicadas (FRöHLICH; Schröder-Preikschat, 2000).

O adaptador de cenário, apresentado na Figura 7, em princípio, se parece com a versão baseada em classes do padrão adaptador (*Adapter*). A abstração independente de cenário é definida por uma interface (*Interface* da Figura 7) para a qual pode haver uma ou mais implementações (classe *Implementor* da mesma figura). A classe *Scenario* é responsável por agrupar os diversos aspectos que serão aplicados na abstração. Esta classe define no mínimo dois métodos, os métodos *enter* e *leave*. Os métodos *enter* e *leave* equivalem aos pontos de corte da AOP, os quais especificam quais pontos de junção do programa terão seu comportamento modificado. No caso, os pontos de junção são, respectivamente, antes da execução do método e depois da execução do método a ser afetado. O código a ser aplicado pelos aspectos (*advice*) é encapsulado em classes de aspectos as quais são utilizadas pelo *Scenario* para implementar seus métodos *enter* e *leave*. A composição da abstração independente de cenário e do cenário é realizada pela classe

ScenarioAdapter, seguindo o padrão apresentado pelo método *operation* da Figura 7, onde a chamada ao método da abstração ocorre entre a chamadas aos métodos *enter* e *leave* do cenário.

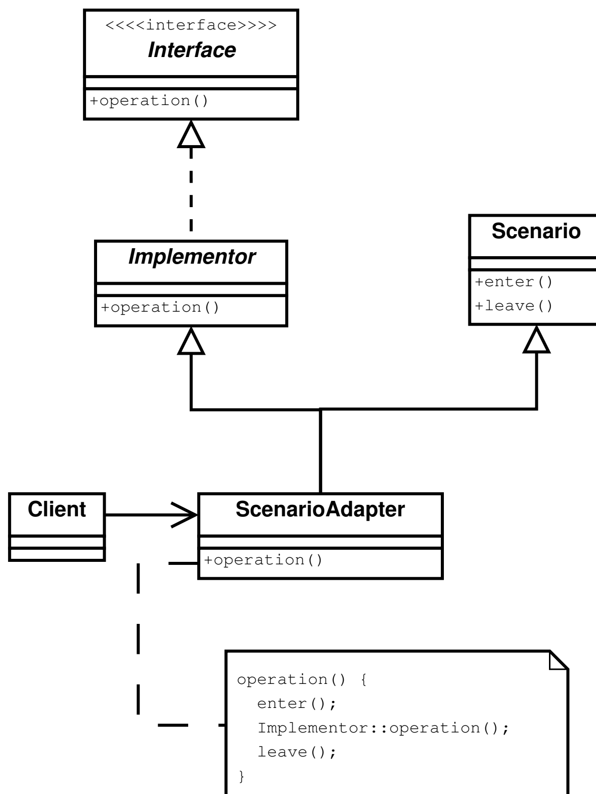

Figura 7: Adaptador de cenário.

Porém, existem importantes diferenças entre os adaptadores baseados em classe e os adaptadores de cenário. Enquanto o adaptador baseado em classes sugere que a amarração (*bound*) entre a classe adaptador (*Adapter*) e as classes adaptadas (*Adaptee*) seja feita em tempo de execução por meio de polimorfismo de subtipo, no adaptador de cenário esta amarração (ocorrida entre *ScenarioAdapter* e as classes *Implementor* e *Scenario*) acontece em tempo de compilação, evitando o custo gerado pela resolução de métodos virtuais. A amarração estática realizada pelo adaptador de cenário não necessariamente implica em perda de flexibilidade. Isto se deve ao fato de que, na maioria das ve-

zes, existe apenas uma implementação (*Implementor*) para uma dada abstração. Independentemente disso, mesmo quando existem mais de uma implementação para uma dada abstração, geralmente os cenários onde estas serão aplicadas são mutuamente exclusivos e a escolha pode ser feita em tempo de projeto. Este é o caso, por exemplo, de uma abstração *Thread*. Esta deve executar em um cenário ou mono-processado ou multiprocessado, e geralmente espera-se que este cenário não mude ao longo da execução da aplicação. Caso os métodos do cenário e do adaptador sejam declarados como *inline*, eles serão dissolvidos na classe cliente que utiliza o adaptador, eliminando-se não somente a chamada a métodos virtuais como qualquer chamada a método, uma vez que na classe cliente a chamada ao método do adaptador será substituída por uma chamada ao método do *Implementor*, entre chamadas aos métodos *enter* e *leave* do *Scenario*.

2.3.3 DERCS

A Engenharia Dirigida por Modelos (do inglês *Model-Driven Engineering* - MDE) propõem o desenvolvimento de sistemas computacionais completos a partir de especificações em mais alto nível as quais são transformadas em uma ou mais etapas para a geração do sistema final. Neste contexto, a especificação Compacta de Tempo Real Embarcada e Distribuída (do inglês *Distributed Embedded Real-time Compact Specification* - DERCS) é uma especificação ou metamodelo utilizado para representar modelos independentes de plataformas. Tais modelos, segundo a metodologia de Engenharia Dirigida por Modelos Orientada a Aspectos para Sistemas Tempo Real (do inglês *Aspect-oriented Model-Driven Engineering for Real-Time systems* - AMoDE-RT) , são utilizados juntamente com a descrição da plataforma alvo e regras de mapeamento para a geração do sistema final, o qual envolve elementos de software e hardware (WEHRMEISTER, 2009).

Na metodologia AMoDE-RT, um modelo DERCS é gerado a partir de diagramas descritos em Linguagem Unificada de Modelagem (do inglês *Unified Modeling Language* - UML) de classe e de sequência e de diagramas que especificam elementos não funcionais na forma de aspectos. Como consequência, o metamodelo DERCS define elementos estruturais, comportamentais e elementos orientado a aspectos, agrupando diferentes visões em um único modelo.

As Figuras 8 e 9 mostram, respectivamente, os elementos estruturais e comportamentais definidos pelo DERCS. Dentre os elementos

estruturais estão classes, atributos, métodos e parâmetros. Os elementos comportamentais determinam o comportamento de um método, detalhando quais as mensagens que este pode passar a outros objetos, quais suas variáveis locais, ações executadas, entre outros. Os elementos dos modelos estruturais e comportamentais do DERCS possuem semântica equivalente à definida em linguagens de programação orientadas a objeto e em UML.

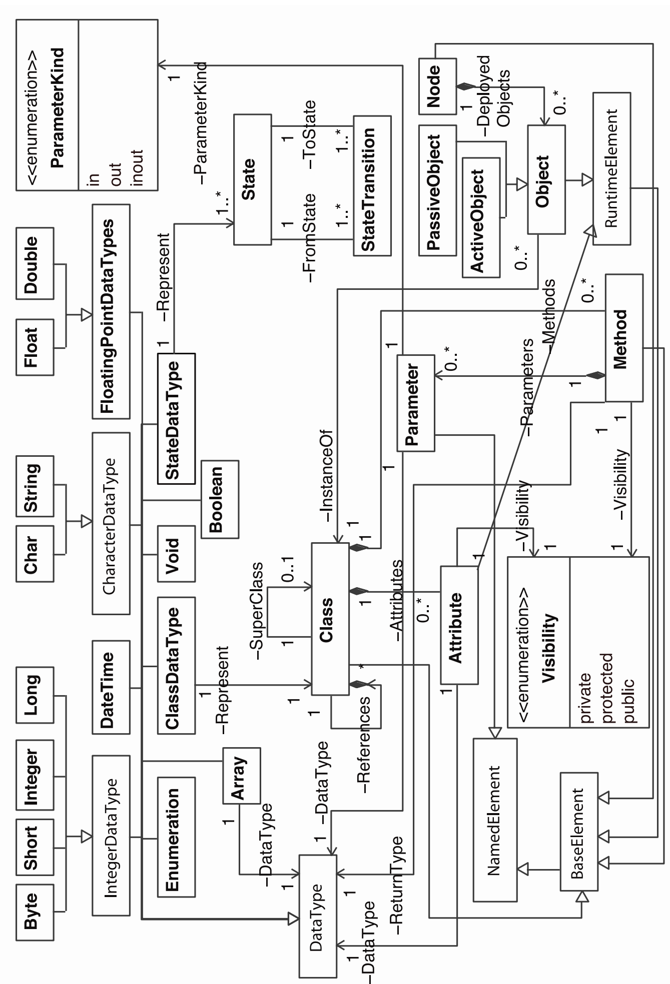

Figura 8: Elementos estruturais do DERCS. Adaptada de (WEHRMEISTER, 2009).

Figura 9: Elementos comportamentais do DERCS. Adaptada de (WEHRMEISTER, 2009).

A Figura 10 apresenta os elementos orientados a aspectos definidos pelo DERCS. Dentre os elementos relacionados a orientação

Figura 10: Elementos de orientação a aspecto do DERCS. Adaptada de (WEHRMEISTER, 2009).

a aspectos estão definidos aspectos, *pointcuts*, *joinpoints* e adaptações estruturais e comportamentais. Estes elementos correspondem aos conceitos de programação orientada a aspectos como definidos em (KICZALES et al., 1997). As adaptações de aspectos (*AspectAdaptation*) representam o conceito de *advices*. As adaptações de aspectos podem ser estruturais (*StructuralAdaptation*) ou comportamentais (*BehavioralAdaptation*). Adaptações estruturais modificam a estrutura dos elementos do modelo, como por exemplo adicionam métodos em uma classe, ou parâmetros em um método. Já as adaptações comportamentais modificam o comportamento dos elementos do modelo, com por exemplo, executando tarefas antes ou depois da execução do comportamento de um método, ou alterando o comportamento de um método por completo.

O metamodelo DERCS pode ser utilizado para modificar a interface de um componente a fim de adaptá-la de acordo com o que é requisitado por um componente cliente, de forma similar ao que é realizado pelo padrão de projeto Adaptador. Esta característica foi

explorada na implementação do método de abstração de componentes de hardware para MPLs proposto nesta dissertação, como é descrito no Capítulo 4.

2.4 LINGUAGEM DE PROGRAMAÇÃO GERENCIADA

Uma Linguagem de Programação Gerenciada (do inglês *Managed Programming Language* - MPL) , é uma linguagem de programação de alto nível que possui funcionalidades tais como o gerenciamento automático de memória, a proteção de memória, e a tipagem forte (BOND; MCKINLEY, 2009; PIZLO et al., 2010; ESMAEILZADEH et al., 2011). Tais funcionalidades facilitam o desenvolvimento de sistemas computacionais de propósito geral, reduzindo o tempo das atividades de codificação e depuração (PHIPPS, 1999). Por meio de proteção de memória, o desenvolvedor pode ser avisado que está tentando acessar uma posição inválida de um vetor ou um objeto não iniciado. O uso de coletores de lixo libera o desenvolvedor da tarefa de escrever código para desalocar objetos e elimina a possibilidade de vazamento de memória. A tipagem forte promove a *segurança de tipo*, impedindo operações entre variáveis de tipos distintos que possam acarretar em resultados inesperados e erros difíceis de detectar. Sob o ponto de vista de MPLs, uma linguagem de programação de alto nível convencional é dita não-gerenciada ou “nativa” sendo referida como Linguagem de Programação Nativa (do inglês *Native Programming Language* - NPL) , ou simplesmente como Linguagem de Programação de Alto Nível (do inglês *High-Level Language* - HLL) .

Duas foram as MPLs escolhidas para a aplicação do método de desenvolvimento de adaptadores de código nativo proposto nesta dissertação: JAVA (ARNOLD; GOSLING; HOLMES, 2005) e LUA (IERUSALIMSCHY, 2006).

JAVA foi escolhida, pois é bem difundida, documentada e possui diversas implementações para trabalhar com sistemas embarcados. A empresa Oracle, proprietária das implementações padrão da linguagem, apresenta versões da plataforma feitas para trabalhar com dispositivos móveis, televisão, e *smart cards* (ORACLE, 2011c). JAVA é uma MPL orientada a objeto que utiliza tipagem estática e explícita. O coletor de lixo de JAVA é capaz de lidar com estruturas circulares de objetos, o que não é tratado por contadores de referência utilizados em diversas MPLs como PYTHON e RUBY. Apesar de não apresentar dicionários e listas embarcadas na sintaxe da linguagem, as bibliotecas fornecidas

para tratamento destas estruturas de dados estão no mesmo nível de abstração das primeiras. Existem também bibliotecas para tratamento de expressões regulares e por meio de IDEs como *Eclipse* e *NetBeans* é possível obter uma execução interativa semelhante aos interpretadores das linguagens que possuem tipagem dinâmica como PYTHON, RUBY e LUA.

LUA é uma linguagem de programação “leve”, multi-paradigma e projetada como uma linguagem de *script* e com de semântica extensível (IERUSALIMSCHY, 2006). LUA apresenta-se como uma boa candidata para desenvolvimento de sistemas embarcados, seu interpretador possui cerca de 150 KB quando compilado, incluindo todas as bibliotecas padrão da linguagem. LUA possui gerenciamento automático de memória, tipagem dinâmica, estruturas de dados embarcadas na própria linguagem e interpretação interativa. A partir da versão 5.1 a linguagem possui coleta de lixo incremental, onde o coletor de lixo executa alternadamente com o programa em execução, escaneando e liberando apenas partes da memória, incrementalmente, a cada vez que é invocado. Por meio da estrutura de dados tabela (*table*), embarcada na própria linguagem, é possível expressar *arrays*, conjuntos, listas e dicionários. O interpretador LUA executa programas LUA em arquivos de *script*, ou podem ser usados de forma interativa, avaliando e interpretando expressões inseridas pelo usuário (LUA.ORG, 2011).

Linguagens de programação de alto nível convencionais, como C e C++, conseguem acessar diretamente dispositivos de hardware. Por meio do conceito de ponteiro é possível armazenar em uma variável, endereços específicos da memória. Manipulando os valores nestes endereços controlam-se os dispositivos de hardware mapeados em memória. Utilizando-se a funcionalidade de *inline assembly* provida pela maioria dos compiladores dessas linguagens é possível especificar em código de montagem (*assembly*) quais instruções devem ser executadas em determinado ponto do programa. Com isso é possível manipular dispositivos de hardware que são controlados por instruções dedicadas, como instruções de entrada e saída.

Entretanto, a manipulação direta e explicita de memória apresenta desvantagens. Todos os objetos alocados dinamicamente precisam ser desalocados explicitamente. A não liberação de um objeto alocado gera o problema de vazamento de memória que pode degradar o desempenho de um sistema ao ponto que sua execução se torne inviável ou gerar erros de difícil detecção.

As MPLs possuem gerenciamento automático de memória. Todos os objetos alocados dinamicamente são desalocados automatica-

mente quando estes não estão sendo mais usados. Um gerenciamento automático de memória facilita a implementação de mecanismos de proteção de memória que, por exemplo, verificam que um objeto que está sendo acessado já foi alocado e ainda possui o seu escopo válido. Por meio de mecanismos de geração e tratamento de exceções é possível acusar violações de acesso a memória por meio de um erro que pode ser facilmente interpretado pelo desenvolvedor do programa.

Entretanto, ao gerenciar a memória de forma automática, o endereço dos objetos é conhecido apenas pelo ambiente de suporte a execução das MPLs, o qual trata de todos os acessos à memória. Isto impede um acesso direto aos dispositivos de hardware, o qual é possível com o uso de linguagens de programação de alto nível convencionais. Para resolver esta limitação as MPLs recorrem um mecanismo chamado de Interface de Função Estrangeira (do inglês *Foreign Function Interface* - FFI) .

FFI é um mecanismo que permite que programas escritos em uma linguagem de programação usem construções de programas escritos em outra linguagem. A linguagem que define a FFI é chamada de linguagem anfitriã (*host language*) e a outra linguagem, a qual tem suas construções utilizadas, é chamada de linguagem convidada (*guest language*). A linguagem anfitriã usualmente é uma linguagem de mais alto nível (i.e. MPL) e a linguagem convidada uma linguagem de mais baixo nível, como C e C++.

Uma FFI pode ser utiliza em duas direções: na primeira direção a linguagem anfitriã usa construções da linguagem convidada, na segunda é a linguagem convidada quem utiliza funcionalidades da linguagem anfitriã. A primeira direção é a mais usual sendo utilizada para realizar acesso a dispositivos de hardware, acesso direto à memória e reuso de código escrito na linguagem convidada. A segunda direção é utilizada para permitir que a linguagem convidada acesse as funcionalidades da linguagem anfitriã de mais alto nível e para embarcar ambientes de suporte a execução das linguagens anfitriãs (e.g. máquina virtuais) em aplicações escritas em linguagem convidada (KORSHOLM; JEAN, 2007). Este trabalho foca na utilização das FFIs na primeira direção, onde a MPL utiliza construções da linguagem convidada para acessar dispositivos de hardware.

As FFIs possuem tal importância para as MPLs que não são utilizadas apenas na construção de novos programas mas também utilizada na implementação das bibliotecas padrão da linguagem. No caso da plataforma *Java Standard Edition* (JSE), pacotes como *java.io*, *java.net* e *java.awt* são implementados utilizando facilidades de FFI. No caso

de Lua o papel da FFI é ainda maior, a *Application Programming Interface* (API) oficial de Lua para interface com C é utilizada para implementar todas as bibliotecas padrão da linguagem, incluindo, manipulação de *strings*, tabelas, funções matemáticas, entrada e saída, interface com sistema operacional e facilidades de depuração.

O Capítulo 3 faz uma revisão das principais FFIs das linguagens Java e Lua e discute a geração automática e depuração de adaptadores de código nativo.

3 AMBIENTES DE SUPORTE À EXECUÇÃO EM MPLS

Este capítulo apresenta as principais FFIs destinadas as linguagens JAVA e LUA e discute a geração automática e depuração de adaptadores de código nativo.

Diversas das FFIs apresentadas foram utilizadas na validação da proposta deste trabalho. Para a linguagem JAVA foram utilizadas as máquinas virtuais KESO e NanoVM. No caso da linguagem LUA, foi utilizada a máquina virtual LUA padrão (LuaVM). Para cada FFI apresentada são descritas as máquinas virtuais nas quais a FFI é empregada e é discutido a adequação da FFI com relação a sistemas embarcados. No caso das máquinas virtuais integradas ao EPOS, discute-se também como o foi realizada esta integração.

Após a apresentação e avaliação das FFIs, é discutido a geração automática de adaptadores de código nativo e a depuração de adaptadores de código nativo, sejam eles gerados de forma manual ou automática.

Por fim, é realizado uma discussão envolvendo as FFIs e geradores de adaptadores de código nativos apresentados destacando as limitações dos mesmos, as quais motivaram a criação do método de abstração de componentes de hardware proposto nesta dissertação.

3.1 INTERFACES DE FUNÇÃO ESTRANGEIRA PARA JAVA E LUA

Na terminologia JAVA FFIs são referidas usualmente como Interface Nativas (do inglês *Native Interfaces*). No caso de LUA a FFI é referida como a API padrão de LUA para interface com C. Na sequencia desta seção são apresentadas FFIs focadas na linguagem JAVA e a FFI padrão de LUA para interface com C. No decorrer da apresentação das FFIs são brevemente apresentadas as máquinas virtuais nas quais essas FFIs são empregadas e é feita uma avaliação da adequação das FFIs apresentadas com relação a sistemas embarcados. No caso das máquinas virtuais KESO, NanoVM e LuaVM, as quais foram integradas ao EPOS, discute-se também como o foi realizada esta integração.

3.1.1 Java Native Interface

Java Native Interface (JNI) é a FFI JAVA utilizada na plataforma *Java Standard Edition* (JSE) (LIANG, 1999). A plataforma JSE é dedicada a computação de propósito geral, sendo a JVM *HotSpot* um de seus componentes centrais.

3.1.1.1 HotSpot VM

A HotSpot é a máquina virtual JAVA padrão da plataforma JSE desde a versão 1.3 da linguagem JAVA. A HotSpot é uma JVM de propósito geral e possui foco em desempenho. Para obter um alto desempenho na execução das aplicações a arquitetura da HotSpot utiliza técnicas de compilação adaptativa, um modelo de *threads* diferenciado das versões anteriores da JVM e um coletor de lixo preciso (*accurate*).

A JVM HotSpot utiliza técnicas de compilação adaptativa como compilação sob demanda (do inglês *Just-In-Time* - JIT *compilation*) e *inlining* de métodos. Diferentemente da técnica JIT clássica, onde o *bytecode* JAVA de um método é traduzido para código nativo a primeira vez em que o método é invocado, o JIT utilizado na HotSpot é controlado por contadores configuráveis de invocação de métodos. Quando estes contadores de invocação ultrapassam certo limite, o método relativo a eles é considerado um *hot spot*, sendo então compilado para código nativo. Ao identificar *hot spots* o compilador da VM tenta executar *inlining* dos métodos que são compilados para código nativo, reduzindo *overhead* de invocação de método causado por troca de contexto.

Diferentemente das chamadas *green threads*, que são threads implementadas completamente pela máquina virtual, a HotSpot implementa as *threads* da linguagem JAVA mapeando-as para as threads do sistema operacional. Neste cenário a responsabilidade do escalonamento das *threads* passa da máquina virtual para o SO; políticas de escalonamento de *green threads*, como a contagem de instruções *bytecode* executadas, são substituídas pelas políticas do modelo de *threads* do SO. Isto não apenas simplifica o projeto e implementação da máquina virtual, como permite tirar vantagem do suporte nativo do sistema operacional e multiprocessamento de forma transparente.

O coletor de lixo da HotSpot é dito preciso (*accurate*) em oposto aos chamados coletores de lixo conservativos (ou parcialmente precisos). Um coletor de lixo conservativo não tem certeza onde estão localizadas todas as referencias a objetos. Desta forma o coletor deve conservati-

vamente assumir que tudo o que se parece com uma referencia a um objeto é de fato uma referência a objeto. Devido a possibilidade de ocorrência de falsos positivos, regiões de memória são impedidas de serem liberadas, o que pode levar ao problema de vazamento de memória (*memory leak*). Por ser preciso o coletor de lixo da HotSpot pode implementar diversas técnicas que permitem uma coleta de lixo mais eficiente (ORACLE, 2011b).

3.1.1.2 JNI

Um dos principais requisitos levados em consideração no projeto da JNI é a compatibilidade binária do código nativo entre diferentes implementações da JVM. Para isto a JNI propõe que o código nativo seja organizado em bibliotecas dinamicamente ligadas, as quais podem ser utilizadas em implementações distintas da JVM sem a necessidade de recompilá-las. A Figura 11 demostra os passos a serem executados na criação de adaptadores de código nativo utilizando a JNI. Uma classe JAVA que declara métodos nativos é compilada com o compilador JAVA padrão. O *bytecode* gerado é analisado pelo programa *javah* que extrai as assinaturas dos métodos nativos e, a partir delas, monta um arquivo de cabeçalho em linguagem C respeitando as regras da API da JNI. Escreve-se então a implementação dos métodos nativos em C e compila-se o código C de forma a gerar uma biblioteca dinamicamente ligável. Finalmente o arquivo de *bytecode* é executado pela JVM a qual, ao verificar a existência de métodos nativos, irá procurar e carregar a biblioteca que os contém. Carregada a biblioteca, a implementação do método nativo pode ser então executada.

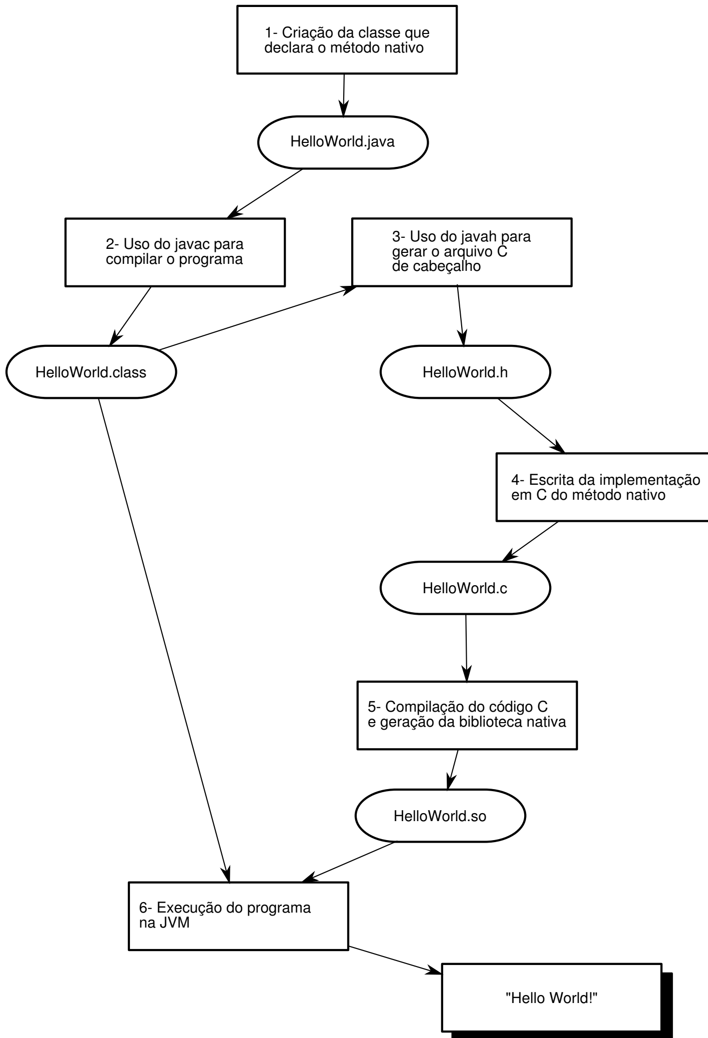

Figura 11: Passos para a criação de adaptadores de código nativo utilizando a JNI.

```
class HelloWorld {
    private native void print ();

    public static void main(String[] args) {
        new HelloWorld().print ();
    }

    static {
        System.loadLibrary ("HelloWorld");
    }

}
```

Figura 12: Classe Java do programa *HelloWorld* utilizando JNI.

```
JNIEXPORT void JNICALL
Java_HelloWorld_print (JNIEnv *, jobject );
```

Figura 13: Arquivo de cabeçalho C gerado pela *javah*.

As Figuras 12, 13 e 14 exemplificam a escrita de adaptadores de código nativo utilizando a JNI. O programa em questão utiliza o método nativo *print* para a escrita da mensagem *"Hello World"* na saída padrão do sistema. A classe Java *HelloWorld* da Figura 12 declara o método *print*. O método *System.loadLibrary* desta mesma classe especifica o nome que deverá possuir a biblioteca contendo o código nativo. As Figuras 13 e 14 correspondem respectivamente aos arquivos de cabeçalho e o arquivo C que implementa o método *print*.

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_print (JNIEnv *env, jobject obj)
{
    printf ("Hello World!\n");
    return;
}
```

Figura 14: Implementação do método *print*.

Como consequência do armazenamento dos métodos nativos em uma biblioteca ligada dinamicamente a interface entre código nativo e JAVA é realizada durante o tempo de execução do programa. Isto significa que, durante a execução de um programa, a JVM procura e carrega a implementação dos métodos marcados como nativos. Este mecanismo de busca e carga de métodos aumenta a necessidade de memória em tempo de execução e o tamanho da JVM. Por esta razão este tipo de mecanismo é evitado em sistemas embarcados.

3.1.2 K Native Interface

A plataforma *Java Micro Edition* (JME) utiliza uma FFI "leve", chamada de *K Native Interface* (KNI) (MICROSYSTEMS, 2002). Esta FFI foi projetada inicialmente para a máquina virtual K - *K Virtual Machine* (KVM).

3.1.2.1 KVM

A KVM é uma JVM criada pela empresa SunMicrosystems a partir da especificação da JVM. O principal objetivo da KVM é prover uma máquina virtual de baixo consumo de memória, na ordem de kilobytes (o "K" do nome KVM significa kilo), com o objetivo de ser utilizada em telefones celulares, *pagers*, organizadores pessoais, entre outros. No projeto da KVM foram considerados além do baixo consumo de memória, desempenho e portabilidade (MICROSYSTEMS, 2000).

Para atingir um baixo *footprint* de memória (aproximadamente 128KB na menor configuração obtida), a KVM impõem algumas restrições a especificação padrão da JVM. Dentre as funcionalidades que podem não estar presentes na KVM estão.

- Tipos grandes de dados, como *long*, *float* e *double*;
- Arrays de mais de uma dimensão;
- Verificação de arquivo *.class* no dispositivo em que eles executarão (esta verificação entretanto pode ser feita previamente *off-line*);
- Não tratamento de alguns erros de classes (a KVM pode optar por simplesmente entrar em estado de *halt*);

- Algumas configurações podem não seguir o modelo padrão de *threads* e tratamento de eventos do Java;
- A JNI não é implementada (ao invés dispõem-se da KNI explicada a seguir);
- Classes podem não ser carregadas dinamicamente;
- Pode não haver suporte para finalização de objetos;
- Pode haver limitações quanto ao tamanho máximo de classes em um pacote, métodos em uma classe, entre outros;
- A carga da classe e execução do método iniciais pode depender da implementação.

Portabilidade é um dos objetivos considerados no projeto da KVM. Para obter portabilidade minimizou-se o número de funções que obtém informação sobre o sistema computacional hospedeiro. O coletor de lixo e o sistema de *threads* foi implementado de forma independente de plataforma. O conceito de *green threads* mencionado anteriormente e a política de escalonamento baseada em número de *bytecodes* executados são utilizados na KVM de modo que o chaveamento de *threads* não dependa de interrupções externas (dependentes de plataforma) e possa ocorrer de forma determinística e fácil de depurar. Esta abordagem é considerada prática pelos projetistas da KVM por não almejar plataformas multiprocessadas (MICROSYSTEMS, 2000). O coletor de lixo da KVM utiliza o algoritmo de "marcar e varrer" (do inglês *mark-and-sweep*) que opera bem em *heaps* de poucas dezenas de kilobytes de tamanho.

A KVM juntamente com um conjunto básico de bibliotecas Java para estruturas de dados, entrada e saída, rede, segurança e internacionalização, definem o chamado *Connected Limited Device Configuration* (CLDC).

As ideias do CLDC e da KVM inspiraram a posterior criação de outras soluções Java para telefones celulares e computadores de mão, como o phoneME (JAVA.NET, 2006) e o Android (GOOGLE, 2008).

3.1.2.2 KNI

A KNI segue a filosofia da KVM, ou seja, baixo consumo de memória mantendo desempenho satisfatório para os dispositivos em que a VM será executada. Para obter um baixo consumo de memória

a KVM “abre mão” da compatibilidade binária do código nativo entre diferentes implementações da JVM, presente na JNI. Desta forma o código nativo deve ser recompilado para cada JVM e alterado caso este dependa do sistema operacional e a JVM em questão objetive outro SO.

A Figura 15 demostra os passos a serem executados na criação de adaptadores de código nativo utilizando a KNI. O primeiro passo é a criação da classe JAVA que declara os métodos nativos. Os métodos nativos são implementados em um arquivo C, seguindo a API da KNI. Estes métodos nativos devem então ser adicionados à tabela de métodos nativos da JVM em questão. Este procedimento pode variar a cada implementação de VM. Um dos procedimentos, por exemplo, é identificar os métodos nativos por números únicos e registrar estes números na tabela de métodos nativos da VM. A JVM deve ser então recompilada a partir dos fontes. Nesta etapa, o arquivo C que implementa os métodos nativos pode ser compilado junto a VM, ou ligado a ela de forma estática. O *bytecode* que corresponde à aplicação JAVA pode continuar a ser gerado por um compilador JAVA padrão. Este *bytecode* deve ou ser carregado pela VM em tempo de execução, ou previamente incorporado à VM (deixando de ser um arquivo *class* padrão) e carregado junto da VM quando esta é executada.

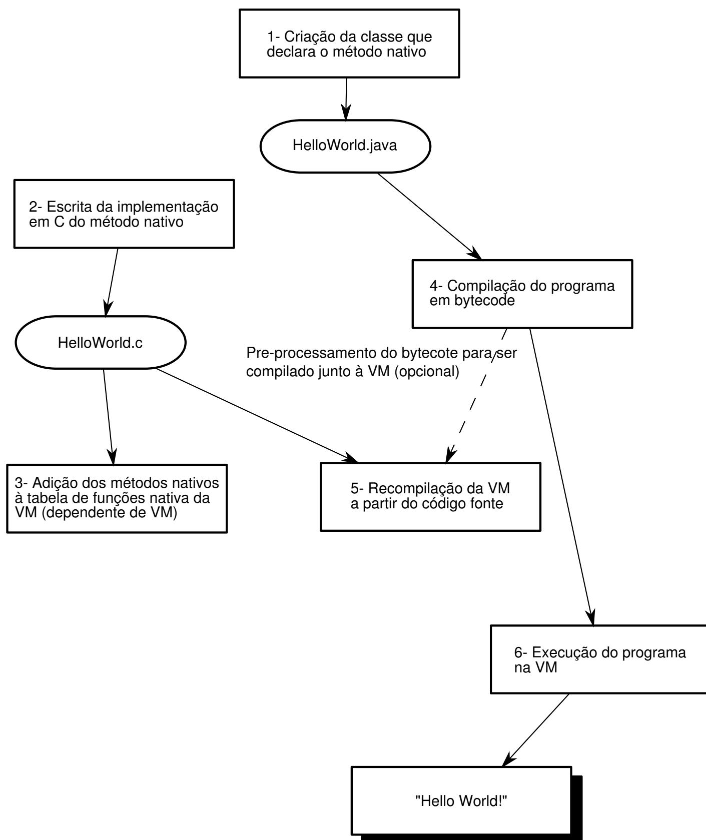

Figura 15: Passos para a criação de adaptadores de código nativo utilizando a KNI.

```
package simplemath;

public class Adder {
    private native void sum(int a, int b);

    public static void main(String[] args) {
        Adder a = new Adder();
        int s = a.sum(2, 3);
        System.out.println("sum: " + s);
    }
}
```

Figura 16: Classe Java do programa Adder utilizando KNI.

```
#include <kni.h>
#include <stdio.h>

KNIEXPORT KNI_RETURNTYPE_VOID
Java_simplemath_Adder_sum() {
     jint a = KNI_GetParameterAsInt(1);
     jint b = KNI_GetParameterAsInt(2);
    KNI_ReturnInt(a + b);
}
```

Figura 17: Implementação do método *sum*.

As Figuras 16 e 17 exemplificam a escrita de adaptadores de código nativo utilizando a KNI. O programa em questão utiliza o método nativo *sum* para calcular a soma de dois números inteiros e retornar o resultado ao método chamador. A classe JAVA *Adder* da Figura 16 declara o método *sum*. A Figura 17 mostra a implementação do método *sum*. A função *KNI_GetParameterAsInt*, provida pela KNI, é utilizada para acessar os argumentos passados a um método. A ordem dos argumentos é a mesma ordem em que os parâmetros formais do método foram declarados, iniciando com o índice 1. Para retornar o resultado da soma ao método chamador (o método *main* do programa JAVA neste caso) a função *KNI_ReturnInt* é utilizada.

A KNI não carrega métodos nativos dinamicamente na JVM, evitando o *overhead* de memória da JNI. Na KNI a interface entre JAVA e código nativo é realizada estaticamente, durante o tempo de compilação. Entretanto decisões de projeto da KNI impõem algumas limitações. A KNI proíbe a criação de objetos JAVA (exceto de strings)

a partir do código nativo. Além disto, na KNI os únicos métodos nativos que podem ser invocados são aqueles pré-compilados na JVM. Não há uma Interface de Programação de Aplicação (do inglês *Application Programming Interface* - API) em nível Java para invocar outros métodos nativos como é o caso da JNI. A KNI é dita uma API de nível de implementação, pois ela é totalmente invisível ao programador Java.

3.1.3 KESO Native Interface

KESO Native Interface (KNI) é a FFI utilizada na JVM KESO. Para evitar confusões com a KNI da Sun/Oracle a *KESO Native Interface* é referida nesta dissertação como FFI da KESO.

Antes de detalhar a FFI da KESO, esta seção faz uma breve revisão da KESO JVM. Em seguida apresenta-se a FFI da KESO e mostra-se como está é utilizada. Após a explicação da FFI da KESO é discutido como a KESO JVM foi integrada com o EPOS, o que permitiu utilizar a FFI da KESO para construção dos adaptadores de código nativo utilizando mediadores de hardware.

3.1.3.1 JVM KESO

KESO é uma máquina virtual Java múltipla (*multi-JVM*) que foca em sistemas embarcados e redes de microcontroladores (WAWERSICH; STILKERICH; SCHRÖDER-PREIKSCHAT, 2007). O termo multi-JVM é devido ao conceito de *domínio* do KESO, pois cada domínio pode ser visto como uma JVM autocontida. KESO baseia-se em sistemas operacionais OSEK/VDX (PORTAL, 2008). OSEK/VDX é uma organização internacional que especifica padrões para sistemas automotivos, incluindo o padrão de sistema operacional utilizado pela KESO. Como a KESO é baseada no OSEK, ela exporta para as aplicações Java os conceitos deste sistema operacional, dentre os quais estão: alocação de tarefas (*tasks*) estaticamente, políticas de escalonamento, mecanismos de interrupção e eventos, e comunicação (GROUP, 2005). A Figura 18, adaptada de (STILKERICH et al., 2006), mostra a arquitetura da KESO e seus principais conceitos.

Tarefas (*tasks*) são as unidades escalonáveis em sistemas OSEK. Elas possuem prioridade fixa e são alocadas estaticamente, durante a geração do sistema. KESO utiliza o conceito de *tasks* OSEK como um substituto para as *Threads* Java. Isto é adequado uma vez que KESO

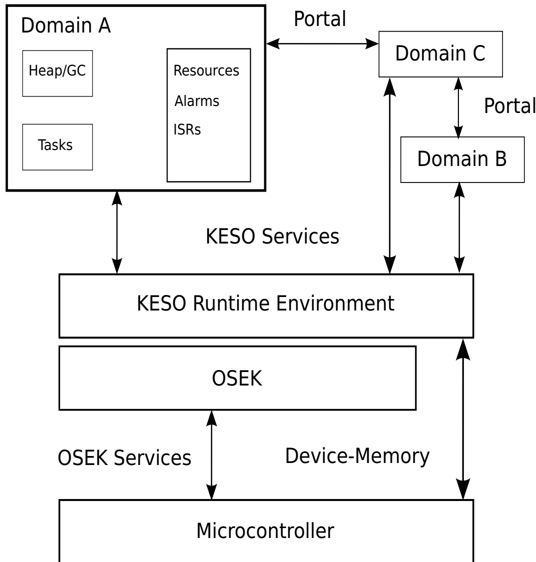

Figura 18: Arquitetura da JVM KESO. Adaptada de (STILKERICH et al., 2006).

foca em sistemas embarcados estáticos, os quais alocam os recursos que serão utilizados pela aplicação durante tempo de projeto.

Apesar de não explorado neste trabalho, a KESO pode ser utilizada como um sistema distribuído composto por *domínios* (*domains*). Um domínio apresenta-se ao desenvolvedor da aplicação como uma JVM autocontida a qual possui sua própria *heap* e campos estáticos de classe. Por causa do conceito de domínio, a KESO é chamada de *multi-JVM*. Uma tarefa no KESO pertence somente a um único domínio e assim como todos os outros tipos de objetos, não pode atravessar as fronteiras de domínios. O conceito de domínio foi introduzido inicialmente pelo sistema operacional Java JX (GOLM et al., 2002) e representa um meio com o qual se pode realizar proteção de memória em software. A utilização de proteção de memória em software é interessante quando a plataforma alvo é desprovida de Unidade de Proteção de Memória (do inglês *Memory Protection Unit* - MPU) e de Unidade de Gerenciamento de Memória (do inglês *Memory Management Unit* - MMU) .

A comunicação inter-domínios é realizada por meio de *portais* (*portals*). Cada domínio pode prover um serviço de portal (*portal ser-*

vice) o qual consiste de uma interface JAVA que oferece serviços (*services*) a outros domínios. Uma tarefa de outro domínio pode importar o serviço oferecido utilizando um serviço global de nomes (*global name service*).

A JVM KESO resolve o *overhead* de interpretação de *bytecode* JAVA compilando-o em código C antes da execução do programa. A Figura 19, adaptada de (WAWERSICH; STILKERICH; SCHRÖDER-PREIKSCHAT, 2007), mostra este processo. Os arquivos fonte JAVA que constituem a aplicação e a biblioteca de classes KESO são compiladas em *bytecode* utilizando-se um compilador JAVA padrão (e.g. *javac*). Então, o compilador do KESO (*KESO Builder*) traduz os arquivos de *bytecode* (.class) em arquivos C e também gera os arquivos C correspondentes às funcionalidades da JVM que serão utilizadas pela aplicação (e.g. coletor de lixo). Durante o processo, o compilador do KESO analisa o *bytecode* e elimina classes, métodos e campos não acessados pela aplicação e elimina, quando possível, chamadas a métodos virtuais.

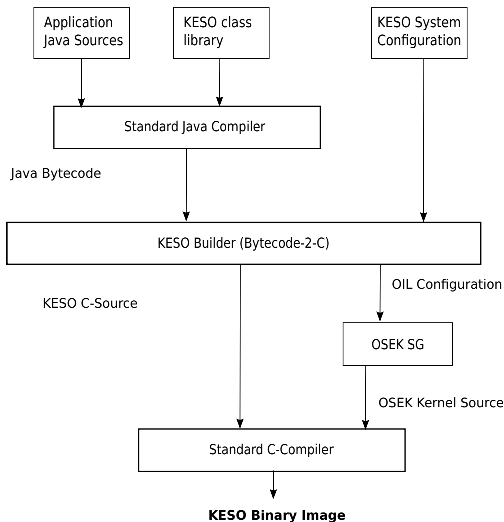

Figura 19: Processo de geração do sistema. Adaptado de (WAWERSICH; STILKERICH; SCHRÖDER-PREIKSCHAT, 2007).

A especificação OSEK afirma que os componentes de sistema operacional utilizados pela aplicação como, por exemplo, tarefas, alarmes e contadores são automaticamente gerados a partir de um arquivo de configuração. Este arquivo de configuração é escrito utilizando-se a

Linguagem de Implementação OSEK (do inglês *OSEK Implementation Language* - OIL) . A KESO utiliza um arquivo de configuração similar, escrito na Linguagem de Configuração KESO (do inglês *KESO Configuration Language* - KCL) , o qual inclui a configuração de domínios, tamanho da *heap* da JVM, além de todas as configurações suportadas por um arquivo OIL convencional. Como mostrado na Figura 19, o compilador KESO lê um arquivo KCL e gera um arquivo OIL a partir do mesmo. Então, o arquivo OIL pode ser utilizado para a geração dos componentes OSEK necessários à aplicação.

3.1.3.2 FFI da KESO

A FFI da KESO utiliza uma abordagem estática assim como a KNI da Sun/Oracle, não realizando carga dinâmica de métodos nativos. Entretanto, diferentemente da KNI, a FFI da KESO provê aos programadores uma API em nível JAVA para criação de novas interfaces com código nativo. Também não existe problema do código nativo chamar código JAVA, uma vez que KESO e a FFI da KESO geram código C.

O projeto da FFI da KESO adota alguns conceitos de AOP. Utilizando a FFI da KESO é possível "escrever" pontos de corte (*point cuts*) especificando os pontos de junção (*join points*) de um programa JAVA (como por exemplo, métodos e classes JAVA) que irão ser afetados pelos conselhos (*advices*) fornecidos. Um *advice*, neste caso, é o código que representa a implementação do um método nativo. A Figura 20 ilustra a utilização da FFI da KESO. Os aspectos (*aspects*), os quais agrupam os *point cuts* e os *advices* são representados na API da FFI da KESO pela classe abstrata *Weavelet*. Estendendo a classe *Weavelet* e implementando alguns de seus métodos, é possível especificar quais métodos e classes JAVA serão afetados e qual código nativo deve ser gerado.

Outro conceito utilizado pela FFI da KESO são os objetos mapeados em memória (*Memory-Mapped Objects*) (THOMM et al., 2010). Um objeto mapeado em memória é um objeto JAVA que, quando criado, pode apontar para um endereço específico de memória do sistema. Estes tipos de objetos são úteis para implementar acesso direto a registradores de hardware.

A FFI da KESO é integrada com o compilador KESO então, durante a compilação de *bytecode* JAVA em C, instâncias de classes *weavelet* são criadas e utilizadas na geração de código nativo. Apesar de que o código especificado por uma *weavelet* não é objeto das análises

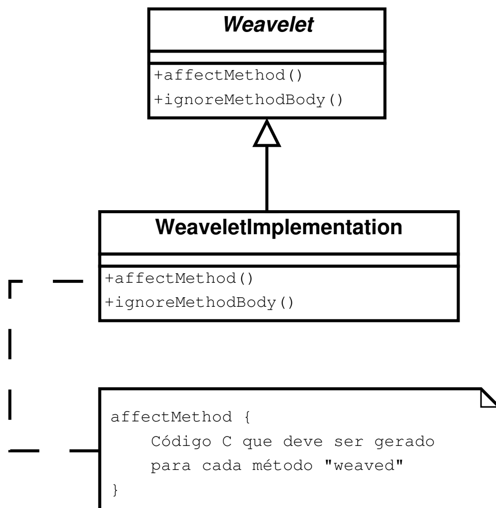

Figura 20: Utilização da FFI do KESO.

estáticas executadas pelo compilador KESO, a FFI da KESO ainda apresenta algumas vantagens interessantes. Por exemplo, se o compilador KESO identifica que o código da aplicação não utiliza algum método nativo, ele não gera o código nativo para aquele método, reduzindo o consumo de memória, o que é altamente desejável em um cenário de sistemas embarcados.

3.1.3.3 Integração com o EPOS

Como mencionado, o compilador KESO gera automaticamente os objetos de sistema operacional a serem utilizados pela aplicação. Dentre estes objetos estão tarefas, contadores e alarmes, os quais são definidos no arquivo KCL em tempo de projeto da aplicação. O código C gerado pelo compilador do KESO contendo tais objetos utiliza a API do sistema operacional OSEK. Logo, uma aplicação desenvolvida com o KESO necessita de um sistema OSEK para sua execução.

Como forma de integrar o KESO ao EPOS, foi proposta por (BAUER, 2008) uma camada que adapta a API do EPOS à API do OSEK. Utilizando esta camada, o EPOS se comporta como um sistema operacional OSEK e pode executar aplicações JAVA geradas pelo KESO.

A Figura 21 mostra o ponto onde a camada desenvolvida, chamada na figura de OSEK/EPOS, se encontra. Faz uso desta camada todo o código gerado pelo KESO correspondente aos objetos de sistema operacional. O código correspondente as classes da aplicação rodam sobre o ambiente de execução do KESO (JVM KESO). O código especificado pela FFI do KESO pode utilizar a camada OSEK/EPOS, ou pode acessar o EPOS diretamente. Os adaptadores de código nativo desenvolvidos nesta dissertação com o uso da FFI do KESO, por exemplo, não necessitam da camada OSEK/EPOS, sendo integrados com os mediadores de hardware disponíveis pelo EPOS de forma direta.

A camada OSEK/EPOS basicamente mapeia construções OSEK em abstrações EPOS, respeitando a semântica OSEK. *Threads* EPOS, por exemplo, são utilizadas na implementação de *Tasks* OSEK de acordo com os estados que uma *Task* OSEK pode assumir e respeitando as politicas de escalonamento definidas por um sistema OSEK.

3.1.4 NanoVM Native Interface

A NanoVM é uma JVM a qual possui como alvo microcontroladores da arquitetura AVR8 (8 bits). A NanoVM especifica e implementa uma FFI própria, referida nesta dissertação como *NanoVM Native Interface*.

3.1.4.1 NanoVM

A NanoVM é um exemplo de máquina virtual dedicada, otimizada para executar em microcontroladores de 8 bits (AVR8) com recursos limitados de memória e processamento. A VM necessita de menos de 8KB de memória de programa e aproximadamente 256 bytes de memória RAM. Para atingir estes valores a NanoVM utiliza-se de técnicas como otimização *off-line* de *bytecode*, além de implementar diversas partes da máquina para executar diretamente sobre hardware (HARBAUM, 2005).

A fim de diminuir o tamanho do *bytecode* e a fim de processá-lo de forma mais rápida, a NanoVM conta com uma ferramenta chamada de *NanoVM tool*. Esta ferramenta possui como entrada os arquivos de *bytecode* compilados pelo compilador JAVA padrão e tem como saída um *array* escrito em C contendo em números hexadecimais o *bytecode* otimizado. Esta otimização é realizada de modo *off-line*, antes do pro-

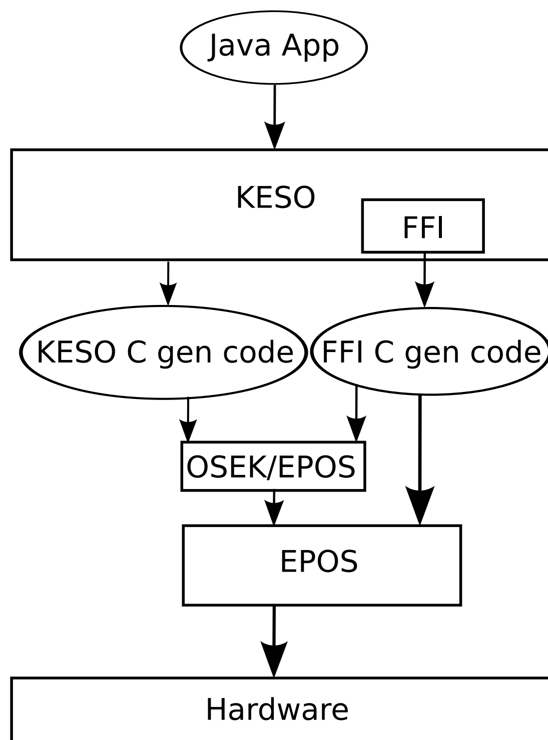

Figura 21: Arquitetura OSEK/EPOS.

grama ser executado pela VM. Dentre as otimizações executadas pela *NanoVM tool* estão: a conversão dos *strings* utilizados para identificar classes e métodos JAVA em identificadores representados por constantes inteiras.

A NanoVM é projetada para rodar diretamente no hardware, sem a necessidade de um sistema operacional. A vantagem desta abordagem é a eliminação do *overhead* de acesso a dispositivos de hardware que um sistema operacional convencional poderia causar. Isto é pago pela total dependência aos dispositivos de hardware presentes nos microcontroladores AVR ATMega8 e ATMega32 o que compromete a portabilidade da NanoVM para outras plataformas de hardware e arquiteturas.

3.1.4.2 FFI da NanoVM

A *NanoVM Native Interface* é similar à KNI em termos de concepção e utilização. A Figura 22 descreve os passos para construção de adaptadores de código nativo utilizando-se a FFI da NanoVM. O primeiro passo é a criação da classe JAVA que declara os métodos nativos. Assim como na KNI, as funções nativas são implementados em arquivos C, os quais devem ser compilados juntos à NanoVM. Para registar os métodos nativos na VM um dos passos necessários é a escrita de um arquivo *.native* o qual especifica as constantes que identificarão o método nativo. Este arquivo é uma das entradas da NanoVM tool, juntamente com o *bytecode* gerado pelo compilador JAVA. O *bytecode* que corresponde à aplicação é otimizado pela NanoVM tool e compilado junto da NanoVM. A NanoVM pode ser então executada carregando a aplicação que foi compilada com ela.

O mesmo programa de soma de números, apresentado na Figura 16 na seção sobre a KNI, pode ser implementado utilizando-se a FFI da NanoVM. Neste caso, o programa JAVA permanece inalterado. A implementação do método nativo que realiza a soma é mostrada no programa da Figura 23 A FFI da NanoVM é bastante rudimentar se comparada com a JNI ou KNI. Como pode se observar, é necessário testar o identificador de cada método nativo da classe *Adder* e as funções *stack_pop_int* e *stack_push* interagem diretamente com a pilha da VM. Além disto, é necessário inserir manualmente uma chamada à função *native_adder_invoke* dentro da implementação da função *native_invoke* da NanoVM a qual, por sua vez, é chamada de dentro do laço principal do interpretador da VM. O registro dos métodos nativos é realizado com a criação do arquivo *.native* e com o registro manual das constantes numéricas que identificam os métodos nativos em um arquivo de cabeçalho da VM.

A ligação entre o código JAVA e o código C é realizada na NanoVM de forma estática, durante a compilação da VM. Além disto, os *bytecodes* da aplicação são pré-processados pela *NanoVM Tool* de forma semelhante ao realizado pelo compilador KESO. Estas características são interessantes para a execução da NanoVM em sistemas embarcados, pois respectivamente, eliminam o *overhead* de carga de métodos nativos e reduzem o custo de interpretação de *bytecode*.

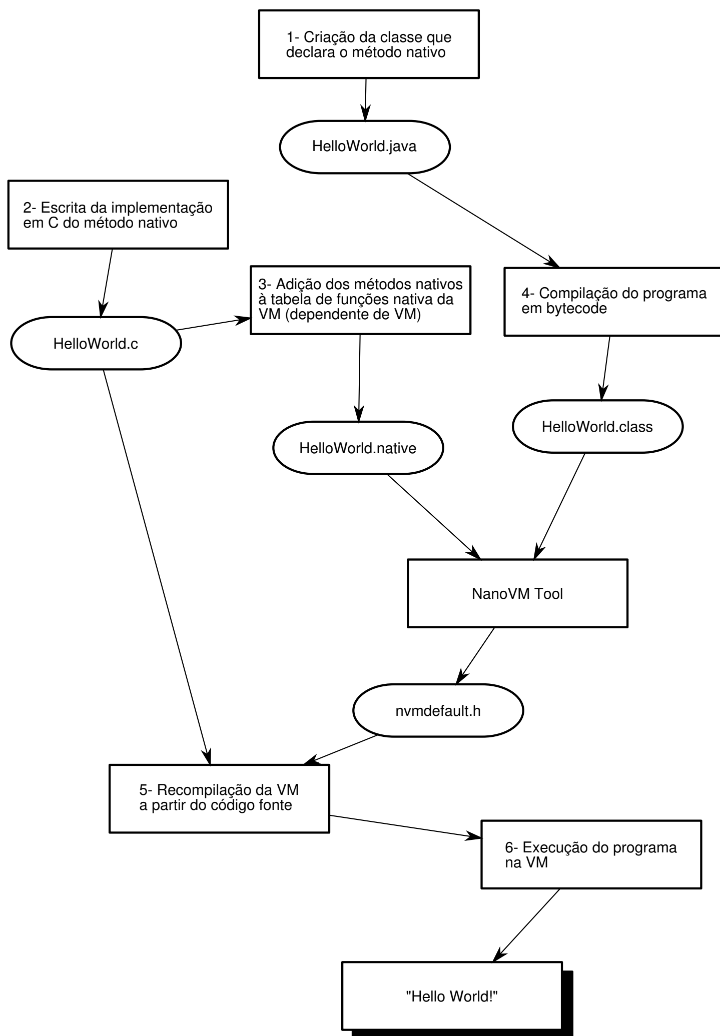

Figura 22: Passos para a criação de adaptadores de código nativo utilizando a *NanoVM Native Interface*.

```
void native_adder_invoke (u08_t mref)
{
    if (mref == NATIVE_METHOD_st_sum) {
        nvm_int_t a = stack_pop_int ();
        nvm_int_t b = stack_pop_int ();
        stack_push(nvm_int2stack(a + b));
    }
    else {
        error (ERROR_NATIVE_UNKNOWN_METHOD);
    }
}
```

Figura 23: Implementação do método *sum* utilizando FFI da NanoVM.

3.1.4.3 Integração com o EPOS

A integração da NanoVM com o EPOS é um trabalho em andamento no Laboratório de Integração de Software e Hardware (LISHA), da Universidade Federal de Santa Catarina.

A estratégia utilizada para integrar a NanoVM ao EPOS consiste em duas etapas. A primeira etapa consiste em identificar e eliminar as partes da VM dependente diretamente de hardware, uma vez que o EPOS fornecerá os mediadores de hardware necessários. A segunda etapa consiste em adaptar a NanoVM para suportar mais de uma *thread* de execução. As *threads* da VM serão então mapeadas às *threads* EPOS.

Apesar da integração da NanoVM com o EPOS ser um trabalho em andamento, a implementação da FFI na máquina virtual se encontra completamente funcional e pode ser utilizada no contexto desta dissertação para o desenvolvimento de adaptadores de código nativo para dispositivos de hardware.

3.1.5 Lua Foreing Function Interface

A FFI de LUA apresenta-se para o usuário por meio da chamada *API C*, a qual é disponibilizada na implementação padrão da linguagem.

3.1.5.1 LVM

O ambiente de suporte a execução de LUA utiliza um esquema de compilação e interpretação semelhante ao utilizado na linguagem JAVA, onde um programa escrito em LUA é traduzido para *bytecode* pelo compilador LUA e então é interpretado pela Máquina Virtual Lua (do inglês *Lua Virtual Machine* - LVM) . A linguagem LUA, incluindo a LVM possui como objetivos de projeto e implementação: simplicidade, eficiência, portabilidade e facilidades para embarcar a linguagem em aplicações C (IERUSALIMSCHY; FIGUEIREDO; CELES, 2005).

Um dos objetivos de LUA é ser uma linguagem simples no sentido de possuir construções sintáticas simples e em número reduzido, o que facilita a implementação e o aprendizado da linguagem. LUA também procura ser simples no sentido de requerer poucas linhas de código C para a sua implementação. Simplicidade de implementação possui impacto também nos outros objetivos da linguagem com eficiência e

portabilidade.

Lua busca eficiência no sentido de compilar e executar programas Lua rapidamente. Compilação eficiente é obtida com uma única varredura no código fonte. O compilador Lua não utiliza representações intermediárias como árvores abstratas de sintaxe e assim que o programa fonte é lido os *bytecodes* correspondentes são gerados. O compilador Lua ocupa cerca de 30% do tamanho do núcleo da linguagem. Para plataformas que possuam limitações de memória é possível eliminar o compilador utilizando diretamente a representação binária de um programa Lua, previamente compilado. Isto contribui também para a execução mais rápida dos programas, uma vez que o tempo de compilação é descartado.

Desde a versão 5.0 da linguagem, Lua possui uma máquina virtual baseada em registradores, ao invés de uma máquina virtual baseada em pilha, como é o caso das máquinas virtuais Java e Python. Um dos argumentos levantados a favor de máquinas virtuais baseadas em registradores é a maior proximidade arquitetural com relação a maioria dos processadores físicos. Em virtude disto, estratégias de compilação sob demanda são beneficiadas e a tradução de *bytecode* da máquina virtual para código nativo do processador físico pode ser feita em menos tempo. Por outro lado, uma máquina virtual baseada em registradores tente a gerar, para um mesmo programa, um código binário maior do que uma máquina baseada em pilha. Isto acontece, pois apesar do número de instruções a serem geradas ser menor, cada instrução em uma máquina baseada em registradores carrega em si a especificação dos seus operandos.

Lua objetiva portabilidade entre diferentes plataformas de software e hardware. Desta forma, o núcleo da linguagem deve ser compilado sem modificações em todas as plataformas e programas Lua devem executar também sem modificações em todas as plataformas que possuam uma implementação da lvm. Isto é obtido com o uso de código C de acordo com o padrão ANSI e sem a utilização de um grande número de bibliotecas C. É objetivo da implementação da linguagem compilar corretamente também em compiladores C++.

Um dos grandes objetivos de projeto da linguagem Lua é o de servir como *linguagem de extensão*, podendo ser "embarcada" em programas maiores, provendo-os com facilidades de uma linguagem de *scripting*, como tipagem dinâmica e sintaxe simples para realização das operações requeridas. Para isto, a linguagem é implementada em sua maioria como uma biblioteca, que pode ser ligada à programas escritos em C. A integração de Lua com C no sentido de prover um programa

C com facilidades Lua é realizada pela API C de Lua. Esta é a mesma API que permite que um programa em Lua acesse funcionalidades de mais baixo nível em C, para a manipulação de dispositivos de hardware.

3.1.5.2 API C

A API C de Lua é bastante similar a KNI do Java em termos de arquitetura e utilização. Desconsiderando as diferenças entre linguagem, os passos necessários para desenvolver adaptadores de códigos nativos são os mesmos da Figura 15 referente à KNI. O primeiro passo é a criação do programa Lua que utilizará as funções nativas. Lua não suporta diretamente o conceito de orientação a objetos, mas utilizando-se das chamadas meta-tabelas de Lua é possível obter em Lua o equivalente a classes e herança simples. As funções nativas não possuem nenhuma marcação especial (como a palavra reservada *native* no caso do Java), entretanto todas as funções nativas devem ser registradas pelo seu nome na LVM. Assim como na KNI as funções nativas são implementados em arquivos C, os quais podem ser compilados juntos a VM, ou ligados a ela de forma estática. O *bytecode* que corresponde à aplicação Lua é gerado pelo compilador Lua padrão o que pode ocorrer durante a execução da LVM ou previamente no caso de não se deseje incorporar o compilador Lua no sistema a ser utilizado.

A LVM é implementada na forma de uma biblioteca em C. É necessário, portanto um programa mínimo em C para instanciar a VM. Este programa, na implementação padrão de Lua é justamente o interpretador interativo da linguagem. A Figura 24 mostra um programa similar ao interpretador do Lua. Por meio da função *lua_open* um *lua_State* é criado. Este representa um ambiente (ou estado) de execução Lua, podendo ser considerado como uma nova instância da LVM. Uma vez instanciada a LVM, são carregadas as bibliotecas padrão de Lua que contém, dentre outras, funções matemáticas e funções de manipulação de *strings*. O programa Lua pode ser então carregado de um arquivo ou de uma *string* armazenada em memória, compilado e executado.

A versão Lua do programa Adder, apresentado na seção sobre a KNI, é mostrado na Figura 25. A função nativa neste caso chama-se *native_sum* a qual é implementada em C, como mostra a Figura 26.
A comunicação entre Lua e C ocorre por meio de uma pilha abstrata, provida pela FFI de Lua. Esta pilha é criada juntamente do estado lua (*lua_State*) quando a LVM é inicializada e é um atributo do mesmo. A

```
int main()
{
    lua_State* L = lua_open();
    luaL_openlibs(L);

    luaL_loadfile(L, "main.lua");
    lua_pcall(L, 0, 0, 0);

    lua_close(L);

    return 0;
}
```

Figura 24: Instanciação da LVM e execução de um programa Lua.

```
Adder = {}
function Adder:new(o)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    return o
end

function Adder:sum(a, b)
    return native_sum(a, b)
end


function main()
    a = Adder:new{}
    s = a:sum(2,3)
    print("sum: ", s)
end


main()
```

Figura 25: Parte Lua do programa Adder.

```
#include <lua.h>
#include <stdio.h>

int native_sum( lua_State* L)
{
    int a = lua_tonumber(L, 1);
    int b = lua_tonumber(L, 2);

    lua_pushnumber(L, a + b);
    return 1;
}
```

Figura 26: Implementação do método *sum* utilizando a API C de Lua.

função *lua_tonumber*, é utilizada para obter um número da pilha abstrata, e a função *lua_pushnumber* é utilizada para colocar o resultado de volta na pilha. Semelhantemente à KNI a ordem dos argumentos obedece a ordem em que os parâmetros formais da função são declarados, iniciando com o índice 1. O registro da função *native_sum* na LVM deve ser feito logo após a criação do estado LUA e carga das bibliotecas LUA padrão, utilizando-se a função *lua_register* da seguinte forma:

```
lua_register(L, "native_sum", native_sum);
```

Esta amarração entre o código LUA e o código C, realizada por meio da função *lua_register*, acontece em tempo de compilação da LVM. Isto é interessante para sistema embarcados, pois se elimina o *overhead* de carregar o código nativo em tempo de execução.

3.1.5.3 Integração com o EPOS

LUA é implementada em ANSI C e as dependências da linguagem com o sistema operacional são resolvidas utilizando-se funções da biblioteca *libc*. Como consequência portar a LUA para um novo sistema operacional implica em resolver estes dependências.

A estratégia utilizada por (MACHADO; FRöHLICH, 2010) para portar LUA para o EPOS foi a de resolver as dependências com a *libc* ou por meio de implementação das funções requisitadas, ou pela substituição de funções da *libc* por outras equivalentes.

Além disto, como o EPOS possui foco em sistemas embarcados, algumas funcionalidades de LUA foram dispensadas como, por exemplo, funcionalidades para manipulação de arquivos e variáveis de ambiente

shell UNIX. Abordagem semelhante foi adotada no projeto *eLua* o qual procura desenvolver uma implementação da linguagem para microcontroladores (PROJECT, 2011).

Foram implementadas funções para alocação de memória, funções de manipulação de *string*, funções para medir tempo transcorrido, geração de números randômicos, saída padrão, entre outras. O perfil completo das funcionalidade de LUA suportadas é apresentado em (MACHADO; FRöHLICH, 2010).

A versão portada inclui a implementação da API C o qual foi utilizada nesta dissertação para construção de adaptadores de código nativo para componentes de hardware.

3.2 GERAÇÃO AUTOMÁTICA DE ADAPTADORES DE CÓDIGO NATIVO

A tarefa de escrita de adaptadores para código nativo pode ser facilitada de duas maneiras, por APIs de alto nível e por ferramentas geradoras. As APIs de alto nível fornecem métodos específicos para auxiliar na criação desses adaptadores, enquanto as ferramentas geradoras podem gerar parte de adaptadores ou adaptadores completos a partir de análise de código nativo ou a partir de uma especificação em mais alto nível.

Simplified Wrapper and Interface Generator (SWIG) e a biblioteca de função estrangeira de PYTHON *ctypeslib* são exemplos de ferramentas que geram adaptadores a partir de arquivos *headers* C/C++ como entrada. O primeiro suporta diversas linguagens como saída como, por exemplo, PYTHON, D e JAVA. O segundo foca em programas PYTHON (SWIG, 2011),(CTYPESLIB, 2011). Ravit et al. apresenta uma ferramenta que tem como objetivo prover funcionalidades da linguagem de mais alto nível (tuplas, por exemplo) para serem utilizadas no código dos adaptadores. A ferramenta proposta por Ravit et al. gera adaptadores PYTHON a partir de código escrito em C e descrições de interface, as quais contêm, dentre outras, informações sobre funções e seus respectivos parâmetros (RAVITCH et al., 2009). Outras soluções, como a linguagem *Jeannie*, misturam código C e JAVA em um único programa a partir do qual geram adaptadores JNI automaticamente (HIRZEL; GRIMM, 2007).

3.3 DEPURAÇÃO DE ADAPTADORES DE CÓDIGO NATIVO

Um adaptador de código nativo pode ser verificado a fim de determinar a presença de erros. Esta verificação pode ser feita estaticamente, a partir da análise do código fonte do adaptador ou dinamicamente, executando-se um programa que use o adaptador e verificando se todas as regras da FFI utilizada são respeitadas.

Ferramentas como J-BEAM e Ilea realizam detecção de erros baseadas na análise de código fonte, utilizando técnicas de análise estática (KONDOH; ONODERA, 2008), (TAN; MORRISETT, 2007).

Lee et al. lidam com detecção dinâmica de erros, no momento em que o código dos adaptadores está sendo utilizado (LEE et al., 2010). A ferramenta deles, Jinn sintetiza detectores dinâmicos de erros para FFIs a partir de máquinas de estado finito, as quais codificam as restrições de FFI que devem ser testadas. A FFI alvo para a linguagem JAVA é a JNI, que contem centenas de chamadas de API.

3.4 DISCUSSÃO

Diversas limitações das FFIs e dos geradores de adaptadores de código nativo motivaram a elaboração do método para abstração de dispositivos de hardware proposto nesta dissertação.

Uma FFI por si só não guia o desenvolvedor na tarefa de abstração de dispositivos de hardware, ela apenas provê meios para que se possa acessar construções de outras linguagens de programação (como C e C++), as quais possuem a capacidade de controlar diretamente dispositivos de hardware (o que é feito por meio de ponteiros e *inline assembly*).

Uma outra limitação no uso manual das FFIs está na suscetibilidade a erros. Diversas FFIs, como KNI, NanoVM FFI e Lua FFI, exigem que o desenvolvedor faça o "*parsing*" de cada argumento do método nativo manualmente (como indicado nas figuras 17, 23 e 26), devendo lembrar-se da ordem em que os parâmetros formais são declarados. Além disto, é necessário que o desenvolvedor tenha em mente as diferenças semânticas entre as linguagens anfitriã e convidada envolvidas pela FFI.

Os geradores de adaptadores de código nativo resolvem o problema de "*parsing*" manual de argumentos e ajudam a contornar as diferenças semânticas entre as linguagens envolvidas pela FFI. A ferramenta SWIG e a biblioteca *ctypedlib* por exemplo, geram adaptado-

res de código nativo a partir de arquivos *headers* C/C++, evitando o "*parsing*" manual de argumentos (SWIG, 2011; CTYPESLIB, 2011). A linguagem *Jeannie* engloba semântica de C e de JAVA e assim possui a capacidade de identificar erros semânticos escritos nesta linguagem, minimizando a geração de adaptadores de código nativo com erros (HIRZEL; GRIMM, 2007). Entretanto, assim como as FFIs, os geradores de adaptadores de código nativo não lidam com o problema de abstração de dispositivos de hardware.

Uma abordagem candidata para resolver o problema de abstração de dispositivos de hardware que não é resolvido pelas FFIs seria por meio do uso de HALs. Porém, como apresentadas no Capítulo 2, as HALs tradicionais não são adequadas para SE, pois encapsulam todos os recursos disponíveis em uma plataforma, representando uma solução monolítica a qual gera interdependência desnecessária entre dispositivos da plataforma de hardware abstraída.

Como detalhado no Capítulo 4, o método proposto para geração de adaptadores de código nativo utiliza o conceito de mediadores de hardware para abstração de dispositivos de hardware em conjunto com FFIs focadas em sistemas embarcados. A geração dos adaptadores é realizada a partir de descrição de mediadores e de descrições de FFIs, sendo que um mediador é automaticamente adaptado à API da FFI alvo como um processo de *weaving* de aspectos.

4 ABSTRAÇÃO DE COMPONENTES DE HARDWARE PARA MPLS EMBARCADAS

Este capítulo apresenta o método para abstração de componentes de hardware para MPLs embarcadas proposto nesta dissertação. Primeiramente, na Seção 4.1, apresentam-se os requisitos que devem ser levados em consideração na elaboração de um método para desenvolvimento de adaptadores de código nativo para controle de dispositivos de hardware. A Seção 4.2 apresenta as ideias que fundamentam o método proposto, mostrando qual a proposta para atender a cada requisito levantado. Finalmente na Seção 4.3 é mostrado o projeto e implementação do método, sendo apresentada também a aplicação do mesmo em Java (JVMs KESO e NanoVM) e em Lua (LuaVM).

4.1 REQUISITOS

Com base nos trabalhos descritos no capítulo 3 e na experiência adquirida com FFIs durante o desenvolvimento deste trabalho, foram identificados quatro requisitos que devem ser atendidos no projeto de um método para realização de interface entre MPLs e dispositivos de hardware. São eles: portabilidade, reuso entre FFIs distintas, desempenho e consumo eficiente de recursos.

1. Portabilidade

   É desejável que a parte do adaptador de código nativo que concentra o código de controle do dispositivo de hardware a ser abstraído utilize interfaces independentes de plataforma. Desta forma, portar o adaptador para outras plataformas significa alterar os métodos que implementam estas interfaces. Nesse caso, o código do adaptador em si permanecerá inalterado.

2. Reuso entre FFIs distintas

   A parte de um adaptador de código nativo que controla o dispositivo de hardware é, em princípio, independente da API da FFI utilizada. Por outro lado, sem o uso dos métodos fornecidos pela API da FFI, não existe interface entre o adaptador de código nativo e a MPL utilizada. A separação entre as partes dependente e independente de FFI permite o reuso do adaptador de código nativo entre FFIs distintas.

3. Desempenho

   Idealmente, o tempo de resposta de um dispositivo de hardware acessado por uma MPL deve ser igual ao tempo de resposta obtido quando utilizando uma linguagem de alto nível com suporte a acesso direto a hardware como C e C++. Para que isto seja verdade, o adaptador de código nativo deve interferir o mínimo possível no tempo de resposta do dispositivo de hardware, caso contrário o seu uso pode tornar-se inviável.

4. Consumo eficiente de recursos

   De forma similar ao desempenho, o consumo de recursos como memória e energia, causado por um adaptador de código nativo deve ser o menor possível para que não impacte no consumo total de recursos do sistema.

4.2 CONCEPÇÃO DO MÉTODO

Tendo como base os requisitos levantados na Seção 4.1, concebeu-se um método de criação de adaptadores de código nativo para abstração de dispositivos de hardware. O método proposto leva em consideração todos os requisitos levantados: portabilidade do adaptador de código nativo entre plataformas distintas, reúso do adaptador de código nativo entre FFIs distintas, desempenho e consumo eficiente de recursos. Questões de tempo real e previsibilidade dos adaptadores de código nativo não foram abordadas na concepção e avaliação deste método. Com relação ao requisito "consumo eficiente de recursos", o recurso mensurado foi memória; não foi avaliado o consumo de energia gerado pelos adaptadores de código nativo.

4.2.1 Portabilidade

Uma das opções para um adaptador de código nativo acessar o dispositivo de hardware a ser abstraído é por meio da utilização do controlador do dispositivo dentro do adaptador de código nativo. Neste caso, os métodos do adaptador de código nativo podem chamar os métodos do controlador do dispositivo ou até mesmo implementar diretamente o controlador do dispositivo. Em ambos os casos, como o controlador de dispositivo é dependente da plataforma de hardware utilizada, o adaptador de código nativo também passa a ser, tendo de

ser reimplementado para cada nova plataforma de hardware utilizada.

Uma possível solução para o problema da portabilidade é a utilização de uma HAL. Neste caso, o adaptador de código nativo implementa seus métodos fazendo uso dos métodos presentes na HAL. Entretanto, como apresentado no Capítulo 2, HALs não são adequadas para sistemas embarcados, pois concentram todos os recursos disponíveis em uma plataforma e geram interdependências desnecessárias entre esses recursos. Desta forma, um adaptador de código nativo que faça uso de uma HAL também conterá abstrações para todos os recursos que a HAL provê, mesmo que não faça uso das mesmas.

A portabilidade de um adaptador de código nativo entre plataformas de hardware distintas é obtida com o uso de mediadores de hardware, apresentados no Capítulo 2. Como existe um mediador de hardware para cada dispositivo de hardware presente em uma plataforma, elimina-se a interdependência entre dispositivos gerada por uma HAL, o que é adequado para sistemas embarcados. Além disso, para cada mediador de hardware a ser utilizado, é desenvolvido um adaptador de código nativo, obtendo-se um controle fino sobre os recursos presentes na plataforma.

A Figura 27 mostra a relação entre um adaptador de código nativo, representado pela classe *Native_Code_Adapter* e um mediador de hardware (*Hardware_Mediator*), para um dispositivo de hardware genérico. O mediador de hardware apresenta uma interface única, para a qual podem existir diversas implementações. Todos os métodos do adaptador de código nativo são construídos utilizando-se esta interface do mediador, portanto o adaptador passa a ser independente de plataforma de hardware.

4.2.2 Reuso entre FFIs Distintas

O reuso de um adaptador de código nativo entre FFIs distintas é obtido por meio da fatoração do mediador em partes funcional (independente de FFI) e parte não funcional (dependente de FFI) e na posterior composição destas duas partes, para a geração do adaptador de código nativo final, o qual será utilizado no contexto de uma FFI específica.

A parte funcional de um adaptador de código nativo é a parte que controla o dispositivo de hardware. Esta parte é dependente apenas do dispositivo a ser abstraído, podendo ser reutilizada em FFIs distintas. Esta parte corresponde aos mediadores de hardware e aos métodos

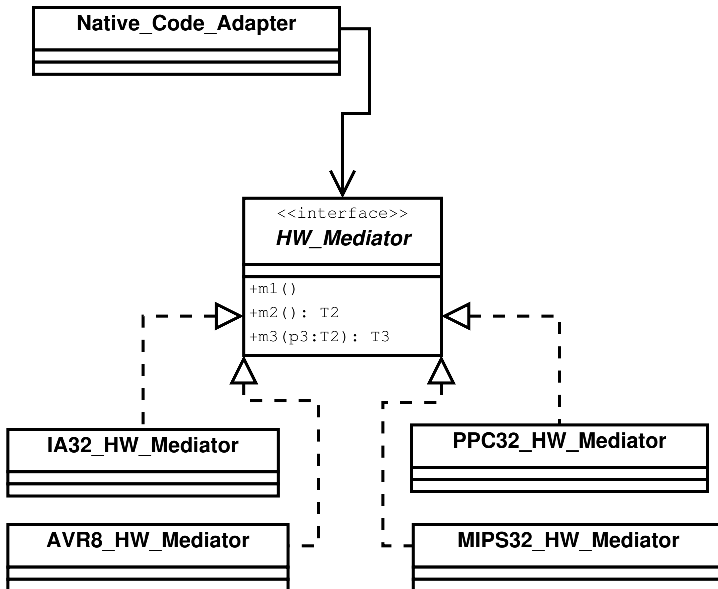

Figura 27: Relação adaptador de código nativo e mediadores de hardware.

presentes nos mesmos.

A parte não funcional de um adaptador de código nativo é a parte dependente da FFI utilizada. Esta parte corresponde ao uso dos métodos dos mediadores de acordo com a API provida pela FFI em questão. Aqui podem ocorrer conversões de tipos envolvendo tipos de dados específicos da API da FFI e os tipos de dados utilizados nos mediadores.

A composição das partes funcional e não funcional de um adaptador de código nativo é encarada como um *weaving* de aspectos da Programação Orientada a Aspectos. Neste caso, a parte funcional dos adaptadores de código nativo, representada pelos mediadores de hardware, é o componente a ser adaptado por aspectos. Os aspectos em si, correspondem à parte não funcional do adaptador de código nativo, contendo todos os detalhes dependentes de FFI.

A Figura 28 mostra a arquitetura geral de uma aplicação em MPL a qual utiliza um objeto MPL, cujos métodos são implementados nativamente. No lado esquerdo da figura são mostradas as partes funcional e não funcional do adaptador de código nativo ainda separadas.

O lado direito da figura mostra o resultado obtido após a aplicação dos aspectos da FFI selecionada. O adaptador de código nativo resultante da composição dos aspectos de FFI e do mediador de hardware contém o mediador de hardware já adaptado para ser utilizado pela FFI escolhida e, portanto já integrado com a MPL utilizada.

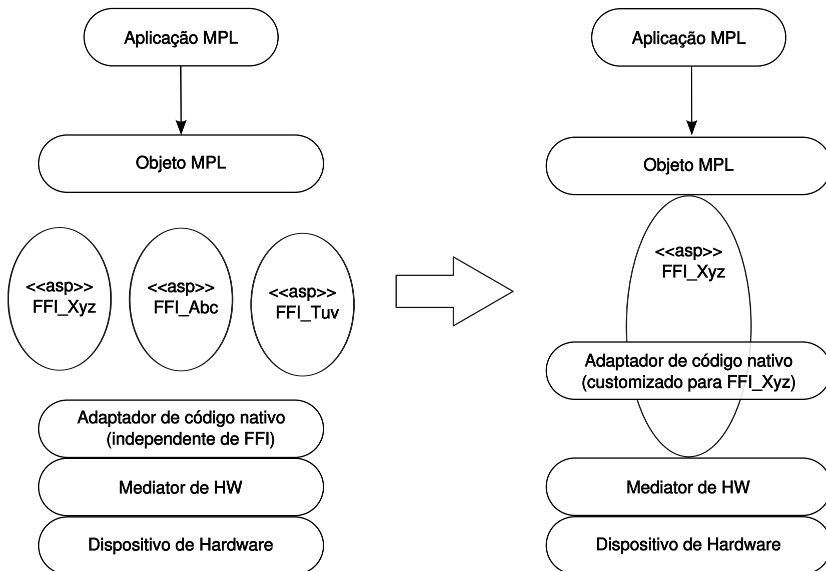

Figura 28: Adaptador de código nativo antes e depois da aplicação dos aspectos de FFI.

4.2.3 Desempenho e Consumo Eficiente de Recursos

O uso de mediadores de hardware permite o desenvolvimento de adaptadores de código nativo com bom desempenho e com uso eficiente de recursos. Um bom desempenho significa que o tempo de resposta obtido no acesso a um dispositivo de hardware por meio do adaptador de código nativo, deve ser próximo ao tempo de resposta obtido quando acessando o mesmo dispositivo de hardware de forma direta, utilizando uma linguagem com suporte direto a hardware como C e C++. O máximo *overhead* de tempo tolerável, causado pelo adaptador de código nativo, é aquele em que todos os requisitos de tempo da aplicação são respeitados. Em uma aplicação tempo real, por exemplo,

isto significa que nenhum *deadline* é perdido. Já no caso de aplicações de alto desempenho, por exemplo, significa manter as taxas de processamento (*throughput*) dentro dos limites esperados. O mesmo vale para recursos como memória e energia, um consumo eficiente é aquele cujos valores possuem a mínima alteração com e sem o uso dos adaptadores de código nativo. Os valores neste caso também dependem da aplicação e não devem interferir nos requisitos de consumo de memória e energia, como por exemplo, tempo de vida de bateria.

São duas as razões pelas quais os mediadores de hardware permitem o desenvolvimento de adaptadores de código nativo com bom desempenho e com consumo eficiente de recursos. A primeira delas está ligada ao projeto dos mediadores de hardware. Como existe um mediador para cada dispositivo presente na plataforma de hardware, o *overhead* causado por falsas interdependências entre dispositivos que ocorre nas HALs é eliminado. A segunda está relacionada às técnicas utilizadas na implementação dos mediadores de hardware. Utilizando-se metaprogramação estática baseada em *templates* e realizando *inlining* dos métodos dos mediadores, estes são dissolvidos no código cliente dos mediadores, eliminando-se o custo de chamadas aos métodos dos mediadores.

No Capítulo 2, os clientes dos mediadores de hardware são as abstrações do EPOS. De forma similar, no contexto desta proposta, as abstrações em MPL são os clientes dos adaptadores de código nativo e dos mediadores de hardware. O nível de desempenho obtido varia de acordo com a abordagem da FFI e da MPL utilizada. Existem dois cenários possíveis, caso a MPL utilize uma representação binária igual à representação binária da HLL nativa (na qual o adaptador de código nativo e os mediadores estão implementados) ou caso a MPL utilize uma representação binária diferente.

A Figura 29 ilustra o caso da MPL utilizar uma representação binária distinta da representação da HLL nativa. Um exemplo deste cenário são as máquinas virtuais JAVA que realizam interpretação de *bytecode*. O *bytecode* é a representação binária dos fontes JAVA (a MPL em questão). A representação binária do mediador de hardware e do adaptador de código nativo escritos em C++ é o código de máquina para uma dada plataforma. Neste caso, podem ser utilizadas técnicas para que os métodos do mediador se dissolvam nos métodos do adaptador de código nativo. No caso de composição do adaptador de código nativo e dos mediadores por meio de herança pode-se dizer que os métodos são os mesmos, uma vez que os métodos do adaptador serão herdados do mediador. Entretanto, como a MPL está representada em

bytecode, existe o custo da chamada ao método do adaptador de código nativo, o qual é invocado pelo objeto em MPL em *bytecode*.

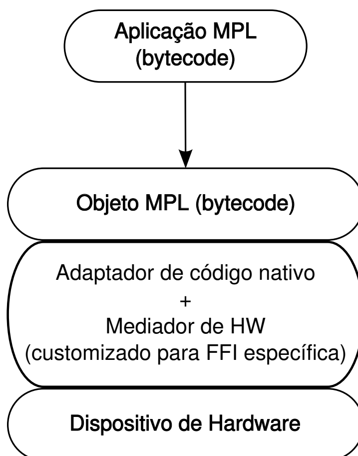

Figura 29: MPL e HLL com representação binária distintas.

O outro caso, representado pela Figura 30, ocorre quando a representação binária da MPL é a mesma que a representação binária do adaptador de código nativo. Este é o caso, por exemplo, de máquinas virtuais JAVA que não realizam interpretação de *bytecode*, mas traduzem o *bytecode* para C, por exemplo, ou diretamente para código de máquina. Neste caso, a representação binária da MPL passa a ser, em última instância, código de máquina, que é a mesma representação binária da HLL (C++) utilizada na escrita do adaptador de código nativo. Nesta situação, é possível dissolver os métodos do adaptador de código nativo diretamente na aplicação que os utiliza. Assim não existe nenhum *overhead* de chamada aos métodos dos adaptadores de código nativo.

Na seção de projeto e implementação do método proposto é detalhado como a composição entre adaptador de código nativo e mediador de hardware pode ser feita.

4.3 PROJETO E IMPLEMENTAÇÃO DO MÉTODO

Esta seção detalha o projeto do método de criação de adaptadores de código nativo para abstração de dispositivos de hardware

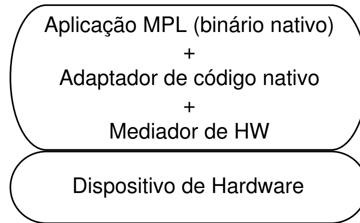

Figura 30: MPL e HLL com a mesma representação binária.

fundamentado nas ideias apresentadas na seção anterior. Esta seção mostra também a implementação desde projeto e sua aplicação nas MPLs Java e Lua. O projeto apresentado resultou na criação de uma ferramenta extensível para geração de adaptadores de código nativo, chamada de *Extensible Binding Generator* (EBG).

A automação da geração de adaptadores de código nativo por meio de uma ferramenta, além de facilitar a construção dos adaptadores de código nativo, evita erros de programação no uso da API da FFI alvo. Como discutido na Seção 3.4, diversas FFIs, como KNI, NanoVM FFI e Lua FFI, exigem que o desenvolvedor faça o "*parsing*" de cada argumento do método nativo manualmente, devendo lembrar-se da ordem em que os parâmetros formais são declarados, além de ter em mente as diferenças semânticas entre das diferentes linguagens que a FFI envolve.

A opção do EBG de utilizar uma representação intermediária com semântica definida, para descrever os mediadores de hardware e aspectos de FFI ao invés de transformações meramente sintáticas sobre o código fonte, permite uma noção de tipos de dados o que contribui para evitar que o código gerado possua erros semânticos.

A Figura 31 apresenta o fluxo completo de geração de um sistema que utiliza o método proposto de interface entre componentes de hardware e MPLs para sistemas embarcados. A partir de uma análise da aplicação, são identificados quais mediadores de hardware serão necessários para suportar a aplicação embarcada. São então selecionados mediadores de hardware existentes, os quais servem de entrada para o EBG. Outra entrada do EBG é a seleção da MPL e da FFI alvo. São saídas do EBG, os adaptadores de código nativo dos mediadores de hardware adaptados para a FFI alvo e a contrapartida em MPL dos mediadores de hardware. A contrapartida MPL dos mediadores é compilada junto com a aplicação por um compilador MPL, o qual irá gerar os *bytecodes* correspondentes. É interessante notar que estes *bytecodes*

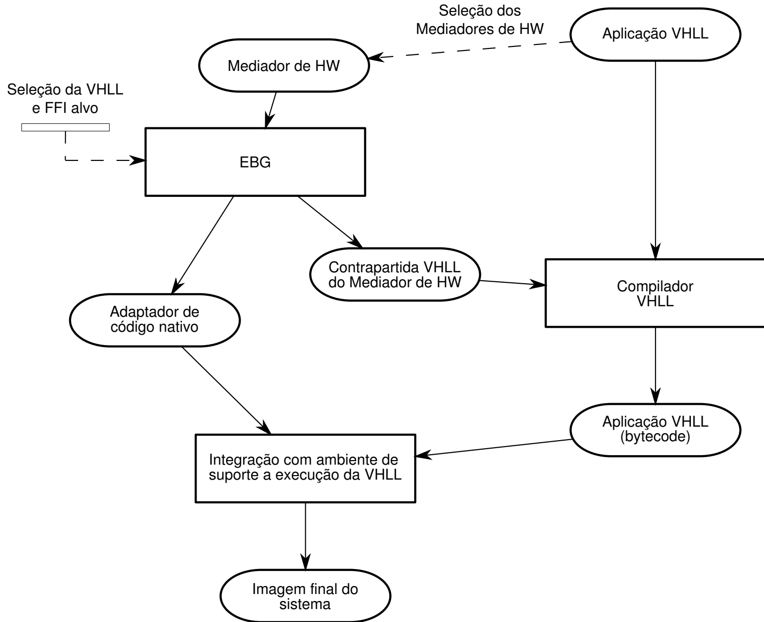

Figura 31: Fluxo completo de geração do sistema.

podem ser externalizados como é o caso dos arquivos *class* em JAVA ou direcionados diretamente ao interpretador da linguagem como é o caso padrão do LUA. Os adaptadores de código nativo gerados pelo EBG e *bytecode* da aplicação são então integrados ao ambiente de suporte a execução (e.g. máquina virtual) da MPL em questão. Este processo é dependente da implementação de MPL utilizada e será discutido caso a caso mais a diante nesta seção. Compilando-se o ambiente de suporte de execução da MPL que inclui o EPOS e os mediadores de hardware, a imagem contendo o sistema final é gerada.

A Figura 32 expande o bloco *EBG* da Figura 31 e mostra como o EBG é organizado internamente. Para cada mediador de hardware que entra no EBG são aplicados os aspectos da FFI selecionada, adaptando o mediador para a API da FFI. A composição do mediador e aspectos é executada pelo módulo *Weaver*. É interessante destacar que os mediadores que entram no EBG, os aspectos de FFI e a saída do *Weaver* em si, são descritos por uma representação interna do EBG que se assemelha muito a uma Árvore Abstrata de Sintaxe (do inglês *Abstract Sintax Tree* - AST) . A saída do módulo *Weaver* entra no módulo *Generator* que transformará a representação interna do EBG para código na linguagem exigida pela FFI da MPL alvo. O código gerado é dependente da FFI alvo e será detalhado mais a diante para cada FFI utilizada neste trabalho.

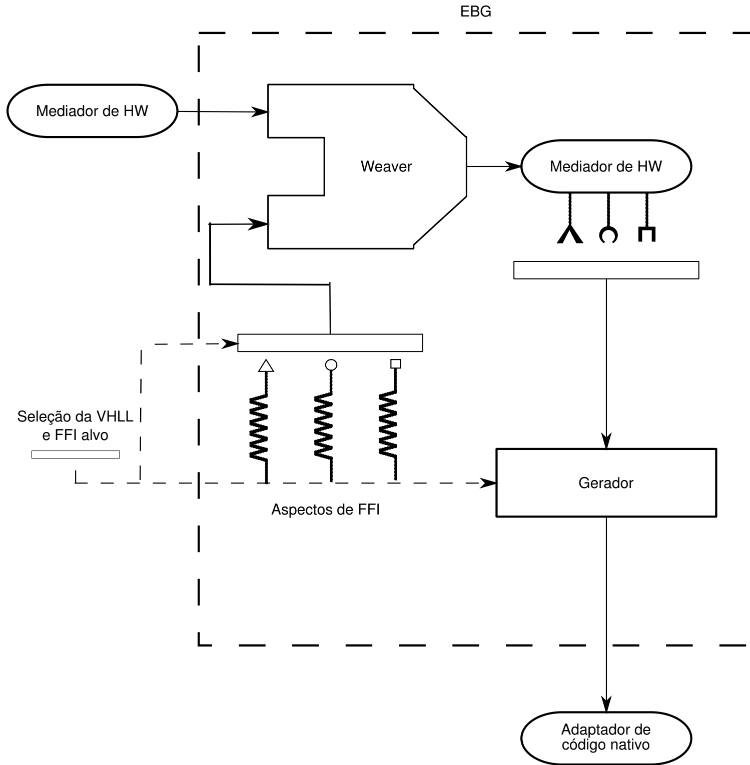

Figura 32: Arquitetura do EBG.

4.3.1 Representação de Mediadores e de Aspectos de FFI

Os mediadores de hardware que entram no EBG estão descritos na linguagem de representação interna do EBG. Esta descrição dos mediadores de hardware em princípio pode ser obtida ou da ferramenta geradora do EPOS, ou ainda a partir da análise dos arquivos de cabeçalho C++ que contém a interface utilizada pelos mediadores.

A representação interna do EBG é baseada no metamodelo DERCS, descrito na Seção 2.3.3. Os elementos estruturais e comportamentais do DERCS são utilizados para descrever os mediadores de hardware. Os elementos do DERCS relacionados à orientação a aspectos são utilizados para descrever os aspectos de FFI. É interessante notar porém que nem todos os elementos do DERCS foram utilizados pois, embora o EBG possa ser integrado com ferramentas de geração de sistemas a partir de modelos que sigam a MDE, isto está fora do escopo deste trabalho.

Outra diferença entre o EBG e a ferramenta *GenERTiCA*, descrita em (WEHRMEISTER, 2009), é quanto à composição de aspectos e elementos estruturais e comportamentais. No caso de (WEHRMEISTER, 2009) como o foco é em MDE, a composição de aspectos ocorre no momento da geração de código para o sistema. O metamodelo DERCS especifica de forma abstrata quais adaptações estruturais e comportamentais devem ocorrer, porém a implementação da semântica destas operações é definida pelas regras que mapeiam o modelo em código. No EBG, a composição de aspectos ocorre totalmente em nível de modelo. Por esta razão neste trabalho estendeu-se o metamodelo DERCS para suportar adaptações estruturais e comportamentais concretas, com semântica definida.

A Figura 33 mostra a extensão da adaptação estrutural *StructuralAdaptation* do DERCS. Definiram-se dois novos tipos concretos de adaptação estrutural, *DeclareParent* e *AddAttribute*. A semântica de *DeclareParent* é definir uma nova superclasse para a classe que será afetada pelo aspecto. Desta forma, a classe alvo desta adaptação passa a ter como superclasse a classe indicada pelo atributo *parent* do metamodelo. O método *weave* presente no metamodelo será explicado quando for comentada a composição de aspectos do EBG. A adaptação *AddAttribute*, quando aplicada, adiciona um novo atributo à classe alvo da aplicação do aspecto.

A Figura 34 mostra a extensão da adaptação comportamental *BehavioralAdaptation* do DERCS. Foram definidos dois novos tipos de adaptações comportamentais, *AddSendMessageAction* e *AddReturnSendMessageAction*. A adaptação *AddSendMessageAction* adici-

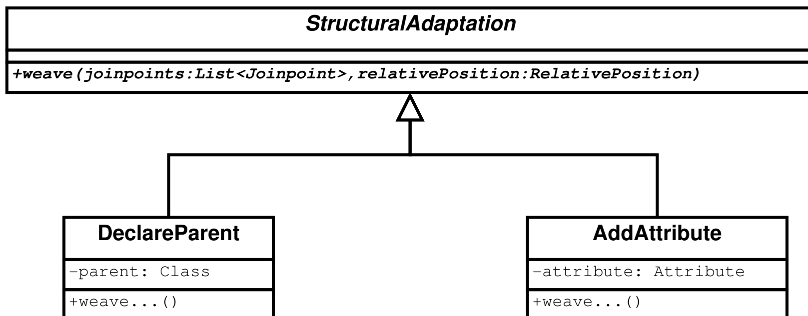

Figura 33: Extensão da adaptação estrutural do DERCS.

ona comportamento ao corpo de um método, mais especificamente ela adiciona uma ação (*Action*) de chamada a um método pertencente a uma classe qualquer, representado pelo elemento comportamental *SendMessageAction*. São especificados por esta adaptação a mensagem a ser adicionada, o método relacionado a esta mensagem e as classes envolvidas. A adaptação *AddReturnSendMessageAction* é idêntica à adaptação *AddSendMessageAction*, exceto pelo fato de que, além de adicionar uma mensagem de chamada a um método, ela considera que esta mensagem irá retornar um valor e que este valor será retornado pelo método que está sendo afetado pelo aspecto. Ambas as adaptações *AddSendMessageAction* e *AddReturnSendMessageAction* podem ser aplicadas no início do comportamento de um método (*BEFORE*), ao final do comportamento (*AFTER*), ou podem substituir o comportamento do método por completo (*AROUND*).

4.3.2 Composição de Mediadores e Aspectos de FFI

O módulo *Weaver* do EBG implementa a ideia apresentada na figura 28, onde são compostos os mediadores de hardware e os aspectos da FFI alvo. O processo de *weaving* no EBG é relativamente simples, sendo apresentado pelo Algoritmo 1. Para cada elemento do modelo que deve ser afetado (i.e. *joinpoint*), o qual é descrito pelos *pointcuts* de um aspecto, aplica-se todas as adaptações de aspectos correspondentes. Da forma como é modelado, um *pointcut* conhece todos os pontos do modelo que ele afetará, assim como conhece todas as adaptações que devem ser aplicadas. Desta forma, a aplicação de uma adaptação de aspecto sobre um elemento do modelo é realizada invocando-se o

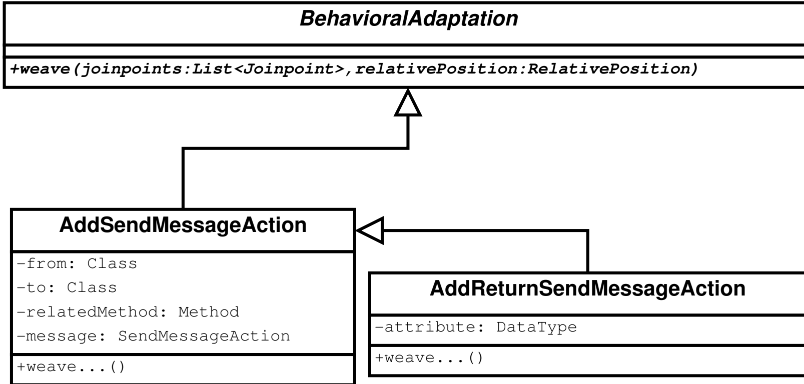

Figura 34: Extensão da adaptação comportamental do DERCS.

Algorithm 1 Processo de composição de aspectos no EBG.

for all $aspects \in model$ **do**
  $aspect \leftarrow model.aspect$
  for all $pointcuts \in aspect$ **do**
    $adaptation \leftarrow pointcut.aspectAdaptation$
    $adaptation.weave(pointcut.joinpoints, pointcut.position)$
  end for
end for

método *weave* que aparece nas Figuras 33 e 34, as quais apresentam as adaptações de aspectos criadas.

No decorrer da elaboração deste projeto foram identificados dois padrões por meio dos quais um mediador de hardware pode ser adaptado à FFI alvo. Estes padrões foram nomeados de Adaptação Baseada em Objeto (do inglês *Object-based Adaptation* - OBA) e Adaptação Baseada em Classe (do inglês *Class-based Adaptation* - CBA) . Estes padrões determinam qual será a relação entre o adaptador de código nativo e o mediador de hardware. No primeiro caso (OBA) o adaptador de código nativo possuirá uma referencia para o objeto que representa o mediador de hardware. No caso de CBA, a composição entre o adaptador de código nativo e o mediador de hardware será feita por meio de herança.

As Figuras 35 e 36 ilustram de forma genérica (independente da FFI alvo) o processo de *weaving* utilizando, respectivamente, os padrões OBA e CBA. Antes de executar os processos de OBA e CBA ou qualquer

tipo de *weaving*, o EBG realiza uma preparação em cima da descrição do mediador de hardware (*HW_Mediator*). A partir de *HW_Mediator*, o EBG gera a contrapartida em MPL do mediador (*MPL_Mediator*) a qual possuirá os métodos presentes no mediador de hardware, porém não possuirá implementação; todos os seus métodos serão marcados como *nativos*. A partir da análise dos métodos de *HW_Mediator*, o EBG gera também um esqueleto de adaptador de código nativo (*NativeMediator*) representado por uma classe com os mesmos métodos de *HW_Mediator* porém também sem implementação.

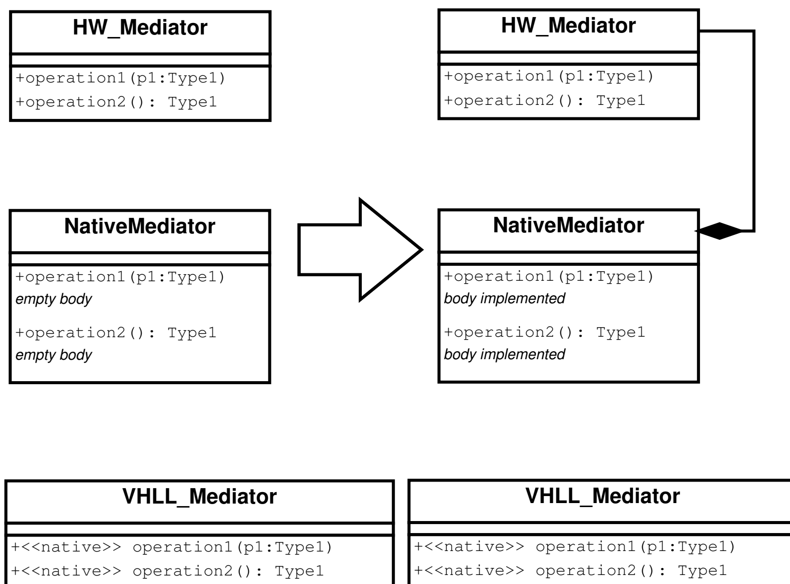

Figura 35: Aplicação de adaptação baseada em objeto.

No caso de OBA, após o *weaving* o adaptador de código nativo *NativeMediator* possuirá uma referencia ao mediador de hardware *HW_Mediator*. Isto é obtido aplicando-se a adaptação estrutural *AddAttribute* em *NativeMediator*. A referencia à *HW_Mediator* será utilizada na implementação dos métodos de *NativeMediator*, utilizando adaptações do tipo *AddSendMessageAction*, já seguindo a interface e especificações da FFI alvo. A Figura 37 ilustra a aplicação do padrão OBA para a FFI da NanoVM. A Figura 38 ilustra o mesmo para a FFI do KESO. Ambas as figuras focam no cenário após a aplicação do *weaving*. Considerando as diferenças dependentes de FFI, pode-se observar

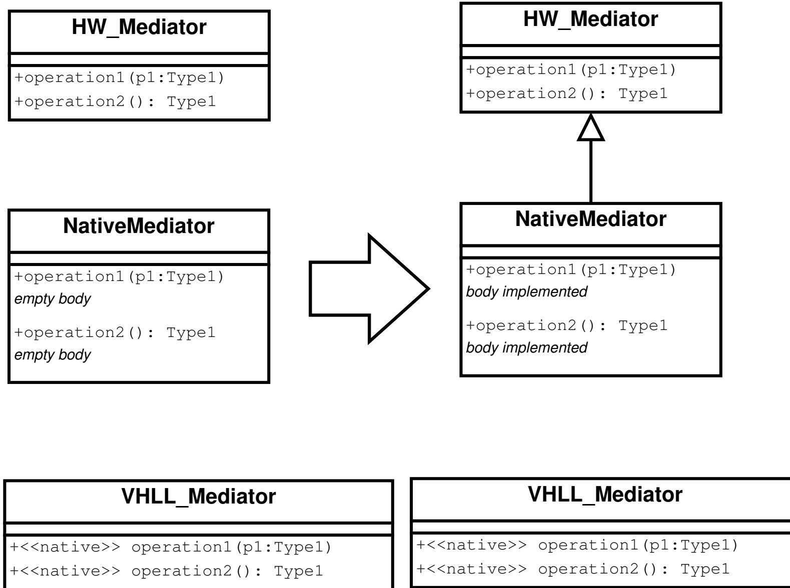

Figura 36: Aplicação de adaptação baseada em classe.

que em ambos os casos o método *operation* do mediador de hardware é invocado por meio da referencia ao objeto *inner* o qual representa o *HW_Mediator* de fato.

No caso de CBA, após o *weaving* o adaptador de código nativo *NativeMediator* estenderá a classe do mediador de hardware *HW_Mediator*. Isto é obtido aplicando-se a adaptação estrutural *DeclareParent* em *NativeMediator*. Utilizando adaptações do tipo *AddSendMessageAction* implementa-se os métodos de *NativeMediator* para a FFI alvo. No caso do uso de CBA, o trecho *inner.operation* das Figuras 37 e 38 é substituído por *operation*, que representa o método herdado de *HW_Mediator*.

4.3.3 Geração Automática de Adaptadores de Código Nativo

A saída do módulo *Weaver* do EBG é um conjunto de adaptadores de código nativo já adaptados para a FFI alvo, mas ainda na linguagem de representação interna do EBG. O módulo gerador do EBG tem como objetivo transformar estes adaptadores em código fonte que

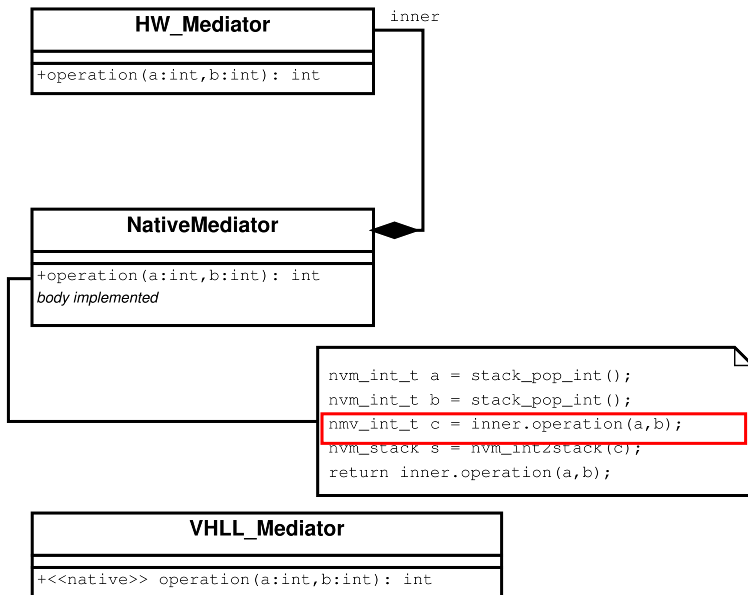

Figura 37: Aplicação de adaptação baseada em objeto para a FFI da NanoVM.

possa ser integrado ao ambiente de suporte à execução da MPL alvo. O procedimento em alto nível desta transformação é mostrado no Algoritmo 2 e consiste basicamente na tradução de corpos de método para a linguagem esperada pela FFI alvo e na aplicação de um ou mais *templates* que formatarão o código de acordo com as regras da FFI utilizada.

Algorithm 2 Geração de adaptadores de código nativo no EBG.

for all $method \in NativeMediator$ **do**
  $methodBody \leftarrow translateToTargetLanguage(method.body)$
  $nativeCodeAdapter \leftarrow applyTargetFFI_Template(methodBody)$
  $write(nativeCodeAdapter)$
end for

No caso da JVM KESO, a saída do módulo gerador do EBG são os arquivos JAVA contendo, respectivamente, a classe *weavelet* e a classe da contrapartida JAVA do mediador de hardware. A classe *weavelet* é utilizada pelo compilador do KESO na geração do sistema final, como

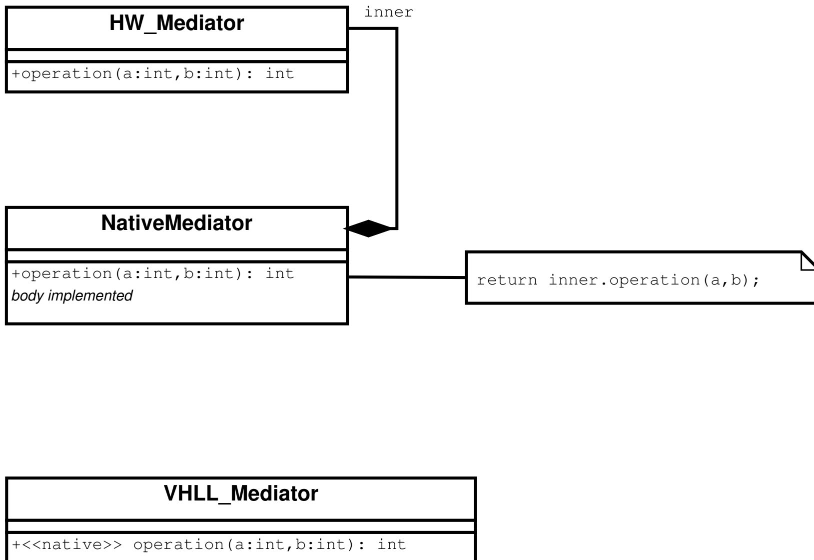

Figura 38: Aplicação de adaptação baseada em objeto para a FFI da KESO JVM.

descrito na Seção 3.1.3.2. A classe da contrapartida JAVA do mediador de hardware é utilizada na aplicação e compilada com um compilador JAVA padrão. Posteriormente, aplicação e classe JAVA do mediador são traduzidos de *bytecode* para a linguagem C, como também descrito na Seção 3.1.3.2.

Como exemplo de código gerado pelo EBG para a FFI da JVM KESO, a Figura 39 mostra os principais trechos da classe *weavelet*. Pode-se observar que o trecho

```
return operation(inner, a, b);
```

foi gerado a partir do corpo do método *operation*, mostrado na figura 38. No caso, a representação interna do EBG foi traduzida para código em linguagem C, especificando exatamente o código que o compilador do KESO deverá gerar. Neste caso específico, como código C foi gerado ao invés de C++, é utilizado um wrapper C/C++ para acessar os métodos do mediador de hardware do EPOS (escritos em C++). Este wrapper "traduz"

```
return operation(inner, a, b);
```

em

```
public class MediatorWeavelet extends Weavelet {
    // ...
    public boolean affectMethod(IMClass clazz, IMMethod method, Coder coder)
throws CompileException
    {
        if (method.termed("operation01(II)I")) {
            coder.addln("return operation(inner, a, b);");
            return true;
        }
        // ...
        return false;
    }
}
```

Figura 39: Classe *weavelet* de um mediador de hardware qualquer.

```
return inner->operation(a, b);
```

É possível observar na Figura 39 o uso do padrão OBA, uma vez que o ponteiro *inner* é um dos atributos do adaptador de código nativo gerado.

5 AVALIAÇÃO DO MÉTODO PROPOSTO

Este capítulo apresenta a avaliação da proposta de interface entre dispositivos de hardware e MPLs embarcadas, introduzida no Capítulo 4, no contexto de JAVA e LUA embarcadas. Primeiramente são apresentadas as métricas utilizadas na avaliação, as quais foram definidas com base nos requisitos apresentados na Seção 4.1. Em seguida apresentam-se as aplicações caso de estudo, nos quais os adaptadores de código nativo para mediadores de hardware foram avaliados e discutem-se os resultados obtidos nas avaliações. A seção final deste capítulo sumariza o que foi aprendido com os casos de estudo e discute contribuições além das originalmente previstas neste trabalho.

5.1 DEFINIÇÃO DAS MÉTRICAS UTILIZADAS

As métricas utilizadas para avaliação das aplicações caso de estudo em relação os requisitos de sistemas embarcados são derivadas do requisitos levantados na Seção 4.1. São elas: desempenho, consumo de memória, portabilidade entre plataformas de hardware e reuso dos adaptadores de código nativo entre FFIs distintas.

5.1.1 Desempenho

Um adaptador que realize interface entre MPLs e dispositivos de hardware deve interferir o mínimo possível no tempo de resposta original de tais dispositivos, caso contrário, sua utilização pode tornar-se impraticável. Como forma de avaliar a proposta do Capítulo 4, foram medidos o tempo de resposta do dispositivo de hardware em análise acessando-o diretamente (e.g. por uma aplicação C++) e acessando-o pelos adaptadores de código nativo desenvolvidos de acordo com a proposta.

O *overhead* de tempo gerado por uma FFI é descrito pela equação 5.1. *Dispositivo* (tempo do dispositivo) é definido como o tempo de resposta original o qual é composto pelo tempo do mediador de hardware mais o tempo do dispositivo físico. *Total* (tempo total) adiciona ao *Dispositivo* o tempo de resposta do método nativo, incluindo a chamada para o método e o retorno do método.

$$OverheadFFI(\%) = \left(1 - \frac{Dispositivo}{Total}\right) \times 100 \qquad (5.1)$$

5.1.2 Consumo de Memória

O adaptador de código nativo que encapsula um dispositivo de hardware deve impactar o mínimo possível na quantidade de memória de código e de dados necessária à execução da aplicação. O *overhead* de memória gerado pelo uso de adaptadores de código nativo é definido pelo aumento no tamanho da área de código e de dados causado pelo adaptador. Uma parcela deste aumento é específica de cada método nativo criado. Outra parcela deste aumento é causada pelo ambiente de suporte a execução da MPL (i.e. VM) e é compartilhada entre todos os métodos nativos.

Como forma de estimar o *overhead* de memória gerado pelo uso de adaptadores de código nativo mediu-se, por método nativo, o aumento no tamanho da área de código e de dados.

Além disso, foi avaliado qual o impacto que todos os métodos nativos em conjunto possuem na aplicação. Para tal, foi comparado a soma do *overhead* de cada método nativo com o *footprint* da imagem binária contendo o sistema como um todo (aplicação e ambiente de suporte a execução), como mostra a equação 5.2.

$$Impacto(\%) = \left(\frac{\sum_{i=1}^{N} Overhead_adaptador_i}{Footprint}\right) \times 100 \qquad (5.2)$$

5.1.3 Portabilidade

Os adaptadores de código nativo foram avaliados quanto à portabilidade de plataforma. Portabilidade de plataforma significa que um mesmo adaptador de código está apto a operar em plataformas distintas de hardware, sem a necessidade de reimplementa-lo para cada plataforma de hardware. Este tipo de portabilidade é obtido construindo-se o adaptador de código nativo sobre interfaces que possuem implementações para diversas plataformas de hardware.

5.1.4 Reuso entre FFIs Distintas

O reuso do adaptador de código nativo entre FFIs distintas é medido pelo número de FFIs distintas que um mesmo adaptador pode ser utilizado sem a necessidade de alterar sua parte funcional (a parte que controla o dispositivo de hardware).

5.2 CASOS DE ESTUDO

Três aplicações foram utilizadas como caso de estudo: uma aplicação de comunicação serial, uma aplicação de estimativa de movimento em codificação de vídeo H.264 e uma aplicação de monitoração de temperatura.

5.2.1 Aplicação Comunicação Serial

O primeiro caso de estudo da aplicação do método proposto nesta dissertação foi o uma aplicação sintética que utiliza um dispositivo de hardware Transmissor-Receptor Assíncrono Universal (do inglês *Universal Asynchronous Receiver Transmitter* - UART) para comunicação serial. Esta aplicação foi avaliada em JAVA, utilizando a FFI KESO, nas arquiteturas IA32 e AVR8. Como indicado pelo nome as duas primeiras são arquiteturas de 32 bits e AVR8 é uma arquitetura de 8 bits.

A versão em JAVA da aplicação, mostrada pelo programa da figura 40, utiliza o mediador de hardware da UART para escrever caracteres em um dispositivo serial. A classe JAVA *UART*, gerada como uma das saídas do EBG, é a contraparte JAVA da classe UART do mediador de hardware e possui apenas métodos nativos sem qualquer implementação. O método utilizado nesta aplicação foi o método *put* e um dos construtores da *UART*. O método *put* possui um único parâmetro, do tipo *char*, que representa o caractere a ser enviado para a UART. O construtor utilizado possui a seguinte assinatura:

```
UART(int baud, int data_bits, int parity,
     int stop_bits, int unit)
```

onde é possível especificar a taxa de transmissão da UART (*baud rate*), bit de dado, paridade, bit de parada e a unidade da UART a ser utilizada, caso a plataforma disponha mais de uma. Existe também outra versão

```
package test;
import keso.core.Task;
import epos.mediador.UART;

public class UART_Test extends Task {
    public void launch() {
        UART serial = new UART(19200, 8, 0, 1, 0);
        while (true) {
            serial .put('M');
        }
    }
}
```

Figura 40: Exemplo UART.

do construtor, sem parâmetros, o qual cria um mediador de UART com o os valores padrão da plataforma utilizada.

Outra saída do EBG é o adaptador de código nativo já adaptado para ser integrado com o ambiente de suporte a execução da MPL alvo. No caso da FFI do KESO isto se traduz na classe *UART_Weavelet* que contém a especificação de implementação de cada método da *UART*, sendo a implementação final gerada pelo compilador do KESO. A Figura 41 mostra a arquitetura geral do programa exemplo UART para a FFI KESO. São mostradas as principais classes envolvidas distinguindo a parte de código JAVA (cinza-escuro) da parte de código C (cinza-claro). Pode-se observar que as classes em JAVA correspondem à quase todo código escrito, incluindo a classe da aplicação *UART_Test*, a classe *UART* do lado JAVA e o *UART_Weavelet*.

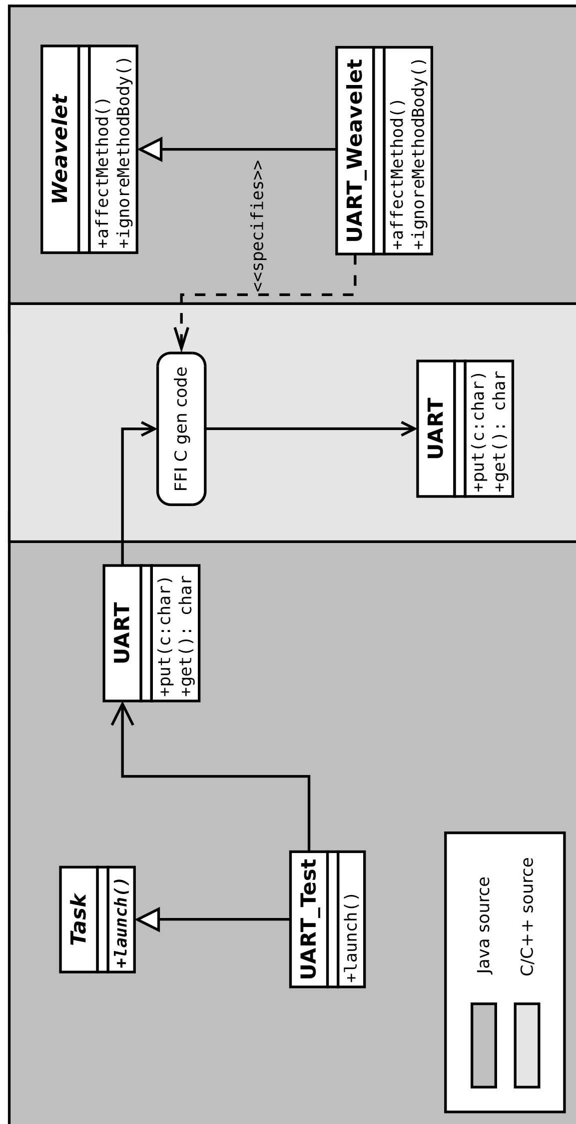

Figura 41: Arquitetura do programa UART para a FFI do KESO.

Tabela 1: *Overhead* de tempo gerado pelo adaptador de código nativo.

	Total (μs)	**Dispositivo (μs)**	**Overhead FFI (%)**
Proposta	517.74	517.54	0.04
JSE	8364683.74	8238695.07	1.5

Com o objetivo de avaliar o desempenho do adaptador de código nativo desenvolvido, foi calculado o *overhead* de tempo para o método *put* da UART na arquitetura IA32 de acordo com a equação 5.1. Foram realizadas 10 amostragens de tempo em 3 execuções distintas da aplicação.

Utilizou-se o *time stamp clock* do EPOS para computar o tempo. O *overhead* obtido, por meio da equação 5.1, corresponde a menos que 0.04% do tempo total de execução do método.

Como forma de estimar a relevância do valor do *overhead*, reproduziu-se o experimento da UART utilizando a plataforma *Java Standard Edition* a qual utiliza a *Java Native Interface* como FFI. Foi utilizada a biblioteca *RXTX* (RXTX, 2011) a qual implementa a *Java Communications API* (ORACLE, 2011a), utilizada para comunicação com dispositivos seriais. A função *gettimeoftheday* foi utilizada para computar o tempo *Dispositivo* e o método JAVA *System.nanoTime* foi utilizado para computar o tempo *Total*. A Tabela 1 mostra os valores obtidos e os compara com os valores obtidos utilizando-se a proposta desta dissertação.

A aplicação baseada em JNI apresenta um *overhead* de 1.5% o qual é aproximadamente 38 vezes maior do que o *overhead* obtido utilizando-se o adaptador de código nativo desenvolvido segundo o método proposto.

O *overhead* de memória da aplicação UART foi medido em todas as arquiteturas para as quais a aplicação foi compilada. A Tabela 2 sumariza o *overhead* causado especificamente pela parte do adaptador de código nativo responsável pelo método *put*. A tabela apresenta também qual o impacto do *overhead* do adaptador de código nativo na imagem final. Como o compilador do KESO gera um arquivo objeto por método, o *overhead* do método *put* foi calculado medindo-se o tamanho destes arquivos objetos com o programa *GNU size*.

Uma vez que um adaptador de código nativo desenvolvido utilizando-se a abordagem proposta utiliza o conceito de mediadores de hardware e este provê uma interface independente de máquina, estes adaptadores

Tabela 2: *Overhead* de memória para *UART::put*.

	Overhead (byte)		**Footprint (byte)**		**Impacto (%)**	
Seção	**IA32**	**AVR8**	**IA32**	**AVR8**	**IA32**	**AVR8**
text	81	144	23741	13172	0.34	1.093
data	0	0	164	160	0	0
bss	0	0	1188	526	0	0
total	81	144	25093	13858	0.32	1.04

podem existir para todas as arquiteturas e plataformas suportadas pelo EPOS. No caso da aplicação da UART as arquiteturas para as quais o adaptador de código nativo foi compilado foram as arquiteturas IA32 e AVR8. O código do adaptador de código nativo é exatamente o mesmo em todas as arquiteturas.

5.2.2 Aplicação Estimativa de Movimento Distribuída

O segundo caso de estudo do método proposto foi uma aplicação real, para a qual se desenvolveu adaptadores de código nativo para um componente que realiza Estimativa de Movimento (do inglês *Motion Estimation* - ME) em codificação de vídeo H.264. O componente em si foi projetado e desenvolvido como parte do trabalho desta dissertação e integrado ao Projeto Rede H.264 SBTVD (H.264, 2009), (LUDWICH; FRöHLICH, 2011c).

O Projeto Rede H.264 tem como objetivo o desenvolvimento de produtos de interesse nacional na área de codificação de sinais-fonte para o Sistema Brasileiro de TV Digital (SBTVD). Uma das questões abordadas pelo projeto é a integração dos seus componentes como por exemplo, dos terminais de acesso interativos que utilizam padrões como o Ginga-J e possuem aplicações escritas em JAVA com os codificadores e decodificadores implementados em C/C++ ou em hardware (GINGA, 2011). Os adaptadores de código nativo propostos e desenvolvidos nesta dissertação permitem que esta integração seja realizada.

O componente proposto realiza a ME, uma técnica utilizada para explorar a similaridade entre imagens vizinhas em uma sequencia de vídeo. A Figura 42 ilustra o processo de ME para duas imagens vizinhas (A e B). Buscando-se semelhanças entre estas imagens é possível determinar onde se encontram na imagem B, blocos da imagem A.

Este deslocamento de blocos é codificado pelos chamados *vetores de movimento*, representado pelas pequenas setas, na parte de baixo da figura. Explorar a similaridade entre imagens vizinhas permite que elas sejam codificadas diferencialmente, aumentando a taxa de compressão do vídeo gerado (WIEGAND et al., 2003).

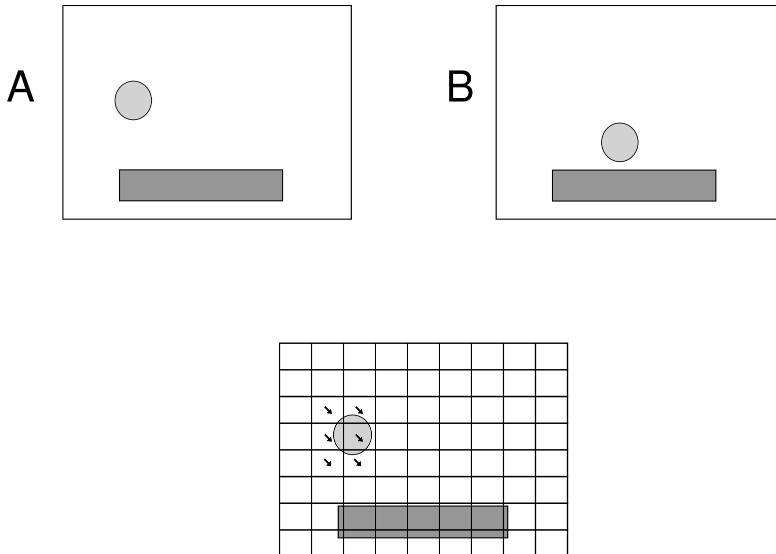

Figura 42: Estimativa de movimento.

ME é um estágio significante da codificação H.264, pois ele consome aproximadamente 90% do tempo total do processo de codificação (LI; LI; CHEN, 2004). Visando aumentar o desempenho da ME, o componente utiliza uma estratégia de particionamento de dados, aonde a estimativa de movimento de cada partição da imagem é realizada em paralelo pelo módulo *Worker*, o qual executa em uma unidade funcional específica, como por exemplo um núcleo de um processador multinúcleo. Existe também o módulo *Coordinator*, o qual é responsável por definir a partição de imagem para cada instância do módulo *Worker* e por prover a eles as imagens a serem processadas. O *Coordinator* é também responsável por agrupar os resultados gerados pelos módulos *Worker* (i.e. custos de movimento e vetores de movimento) e por entregar estes resultados de volta para o codificador. A Figura 43 ilustra a interação entre os módulos *Coordinator* e *Worker*.

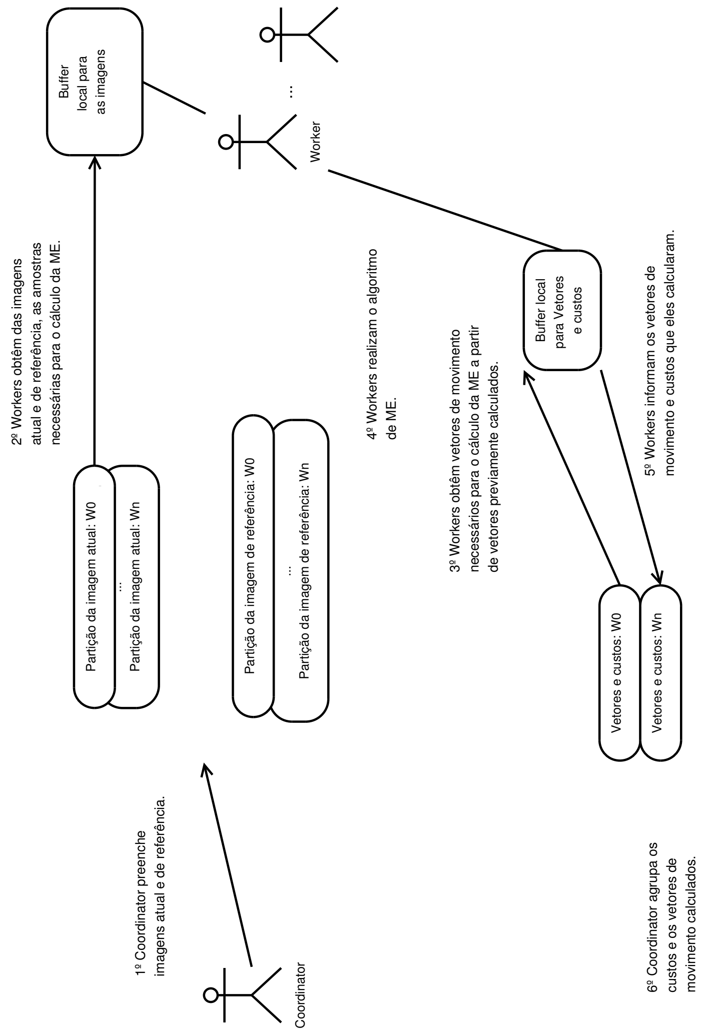

Figura 43: Interação entre *Coordinator* e *Workers*.

```
/* pme_native_match(self.inner, currentPicture.inner, referencePicture.inner) */
int pme_native_match(lua_State* L)
{
    PictureMotionEstimator* pme =
        static_cast <PictureMotionEstimator*>(lua_touserdata(L, 1));

    MEC_Picture* currentPicture =
        static_cast <MEC_Picture*>(lua_touserdata(L, 2));

    MEC_Picture* referencePicture =
        static_cast <MEC_Picture*>(lua_touserdata(L, 3));

    PictureMotionCounterpart* pmc;
    pmc = pme->match(currentPicture, referencePicture);

     lua_pushlightuserdata (L, static_cast <void*>(pmc));

    return 1;
}
```

Figura 44: Adaptador de código nativo para o método *match* (API C de Lua).

O componente é chamado de Componente de Estimativa de Movimento Distribuída (do inglês *Distributed Motion Estimation Component* - DMEC) , uma vez que o cálculo da ME é realizado de forma paralela pelos módulos *Worker*. Entretanto, esta complexidade é ocultada da aplicação (e.g. codificador H.264), o qual enxerga apenas um componente para calculo de ME, realizada pelo método *match*. O DMEC é implementado como um componente em C++ e é exportado para a MPL alvo desenvolvendo-se um adaptador de código nativo para cada objeto sendo abstraído.

Utilizando-se o EBG foram gerados adaptadores de código nativo para o DMEC trabalhar com Java e em com Lua. A Figura 44 mostra o adaptador de código nativo para o método *match* do DMEC gerada para a FFI de Lua. A Figura 45 mostra o equivalente para Java, utilizando-se a FFI do KESO. No caso do Lua, o adaptador gerado já está pronto para ser utilizado. No caso do Java, o adaptador gerado é uma classe *Weavelet*, a qual é usada no processo de compilação da KESO, gerando o código C do adaptador final.

```
public class OBA_C_PME_Weavelet extends Weavelet {
    // ...
    public boolean affectMethod(IMClass clazz, IMMethod method, Coder coder)
throws CompileException
    {
        if (method.termed(
"match(Ldmec/Picture;Ldmec/Picture;)Ldmec/PictureMotionCounterpart;"))
        {
            IMMethodFrame frame = method.getMethodFrame();
            IMSlot[] arguments = frame.getMethodArguments();
            assert(arguments.length == 3);
            IMSlot thiz = arguments[0];
            IMSlot currentPicture = arguments[1];
            IMSlot referencePicture = arguments[2];

            String _thiz =
                WeaveletUtility.mountArg("PictureMotionEstimator*", thiz);
            String _currentPicture =
                WeaveletUtility.mountArg("MEC_Picture*", currentPicture);
            String _referencePicture =
                WeaveletUtility.mountArg("MEC_Picture*", referencePicture);

            String _args = _thiz + ", " + ", " + _currentPicture + ", " +
                _referencePicture;
            coder.addln(_thiz + "pme_match(" + _args + ");");
            coder.addln("return " + thiz + "->__pmc;");

            return true;
        }

        // ...


        return false;
    }
}
```

Figura 45: Adaptador de código nativo para o método *match* (FFI do KESO).

```
public class DmecApp extends Task {
  public void launch() {
    DebugOut.println("DMEC␣APP␣is␣alive!");
    int width = 1920;
    int height = 1088;
    PictureMotionEstimator pme = new PictureMotionEstimator(width, height);

    Picture currentPicture = TestSupport.createPicture(width, height, 0);
    Picture referencePicture = TestSupport.createPicture(width, height, 1);

    PictureMotionCounterpart pmc = pme.match(currentPicture, referencePicture);

    TestSupport.testPMC(pmc, width, height, currentPicture, referencePicture);
  }
}
```

Figura 46: Aplicação Java DMEC.

Tabela 3: *Overhead* de tempo gerado pelo adaptador de código nativo de *DMEC::match*.

Total (μs)	**Dispositivo (μs)**	**Overhead FFI (%)**
971615.9084596	971615.9	0.0084596

Foi escrita uma aplicação para testar os adaptadores de código nativo gerados. A Figura 46 mostra a versão em Java da aplicação e a Figura 47 mostra a versão em Lua. Do ponto de vista da ME, a aplicação desenvolvida atua como um codificador H.264: ela provê ao DMEC imagens para serem processadas e utiliza os resultados entregues pelo componente verificando se eles estão corretos.

O desempenho dos adaptadores de código nativo para o DMEC foi avaliado de acordo com a equação 5.1. A Tabela 3 mostra os valores obtidos para a FFI da KESO. Para avaliar o consumo de memória causado pelos adaptadores de código nativo, mediu-se o *overhead* de memória causado pelos mesmos. A Tabela 4 mostra o *overhead* causado pelo adaptador de código nativo responsável pelo método *match*, utilizando-se a FFI da KESO. A tabela apresenta também qual o impacto do *overhead* dos adaptadores de código nativo na imagem final.

Foram gerados adaptadores de código nativo para todos as classes que envolvem o DMEC, incluindo classes para representação das imagens (*Picture*), estimativa de movimento (*PictureMotionEstimator*) e retorno da ME (*PictureMotionCounterpart*). Foi gerado também um

```
function launch()
    print("DMEC APP for KESO is alive!");

    pictureWidth = 1920
    pictureHeight = 1088
    pme =
        PictureMotionEstimator:new{width = pictureWidth, height = pictureHeight}

    currentPicture = createPicture(pictureWidth, pictureHeight, 0)
    referencePicture = createPicture(pictureWidth, pictureHeight, 1);

    pmc = pme:match(currentPicture, referencePicture);

    testPictureMotionCounterpart(pmc, pictureWidth, pictureHeight,
        currentPicture, referencePicture);
end

launch()
```

Figura 47: Aplicação Lua DMEC.

Tabela 4: *Overhead* de memória para *DMEC::match*.

Seção	**Overhead (byte)**	**Footprint (byte)**	**Impacto (%)**
text	201	57989	0.35
data	0	1424	0
bss	0	1405556	0
total	201	1464969	0.01

adaptador de código nativo para a classe de suporte aos testes (*TestSupport*) do DMEC, porém este não foi incluído nas medições de *overhead* pois não está relacionado ao calculo da ME, apenas do testes destes resultados, simulando um codificador de vídeo.

5.2.3 Aplicação de Monitoração de Temperatura

O terceiro caso de estudo foi uma aplicação de monitoração de temperatura, ilustrada na Figura 48. Trata-se de uma aplicação distribuída, constituída de um nodo que realiza medições de temperatura (nodo *sensor*) e envia os valores obtidos para um nodo que processa as informações recebidas (nodo *sink*). A comunicação entre os nodos acontece por meio de rádio, no contexto de uma Rede de Sensores em Fio (RSSF).

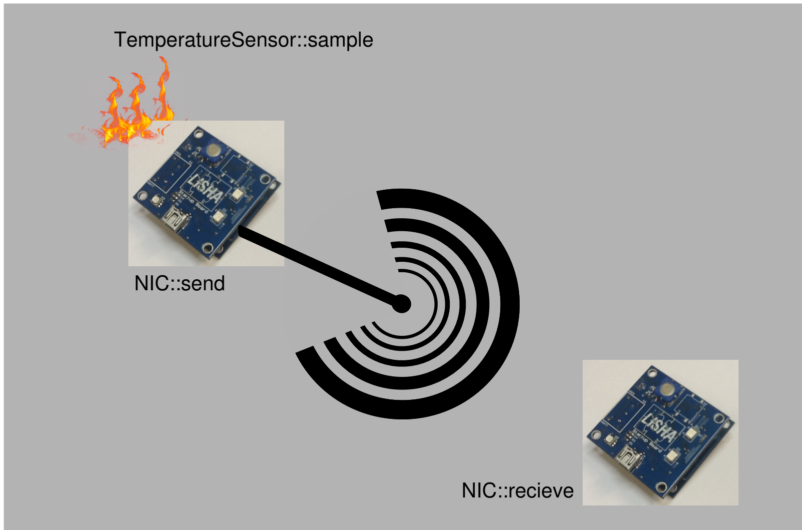

Figura 48: Aplicação de monitoração de temperatura.

A Figura 49 mostra a aplicação que é executada no nodo *sensor*, enquanto a Figura 50 mostra a aplicação que executa no nodo *sink*. A aplicação foi escrita em JAVA e foram gerados adaptadores de código nativo para os mediadores sensor de temperatura (*Temperature_Sensor*) e interface de rede NIC, que abstrai o rádio utilizado para comunicação entre os nodos. As FFIs alvo foram FFI da KESO e FFI da NanoVM e em ambas as máquinas virtuais a aplicação é a mesma.

```java
package temperature_sensing_app;

import epos.mediator.NIC;
import epos.mediator.TemperatureSensor;
import epos.abstraction.PeriodicThread;

public class SensorApp extends PeriodicThread {

    final int DATA_SIZE = 32;
    private char msg[];
    private NIC nic;
    private TemperatureSensor temp;
    private char c;

    public SensorApp(int period) {
        super(period);
        nic = new NIC();
        msg = new char[DATA_SIZE];
        temp = new TemperatureSensor();
        c = 0;
    }

    public void run() {
        char id = 1;
        System.out.println("Sensor id = " + id);

        for (int i = 1; i < DATA_SIZE; i++) {
            msg[i] = (char) i;
        }
        msg[0] = id;
        msg[1] = c++;
        msg[2] = temp.sample();

        int r;
        if ((r = nic.send(NIC.BROADCAST, msg)) != NIC.SENT_OK) {
            System.out.println(" failed " + r);
        }

        System.out.println("tx done");
    }

    public static void main(String[] args) {
        int period = 1000000; // 1s
        SensorApp sensor = new SensorApp(period);
        sensor.start();
    }
}
```

Figura 49: Aplicação *sensor*.

```
package temperature_sensing_app;

import epos.mediator.NIC;
import epos.abstraction.PeriodicThread;

public class SinkApp extends PeriodicThread {

    final int DATA_SIZE = 32;
    private char msg[];
    private NIC nic;

    public SinkApp() {
        nic = new NIC();
        msg = new char[DATA_SIZE];
    }

    public void run() {
        System.out.println("Sink");

        if (!(nic.receive(msg)) > 0)) {
            System.out.println(" failed \n");
        }

        System.out.println("Sender id: " + msg[0]);
        System.out.println("Msg number: " + msg[1]);
        System.out.println("Temperature: " + msg[2] + " C");
    }

    public static void main(String[] args) {
        int period = 1000000; // 1s
        SinkApp sink = new SinkApp(period);
        sink.start();
    }
}
```

Figura 50: Aplicação *sink*.

Tabela 5: *Overhead* de tempo gerado pelos adaptadores de código nativo. Arquitetura AVR8, FFI da KESO.

Método	**Total (μs)**	**Disp. (μs)**	**Overhead FFI (%)**
Temp::sample	334.22	330	1.26
NIC::send	8586.22	8580	0.072

Tabela 6: *Overhead* de tempo gerado pelos adaptadores de código nativo. Arquitetura ARM7, FFI da NanoVM.

Método	**Total (μs)**	**Disp. (μs)**	**Overhead FFI (%)**
Temp::sample	946.04	942	0.43
NIC::send	958.39	950	0.87

A aplicação que executa na KESO JVM foi executada na arquitetura AVR8 (8 bits), a aplicação que executa na NanoVM foi executada na arquitetura ARM7 (32 bits). A plataforma utilizada em ambos os casos foi o *EPOS Mote* (versões AVR8 e ARM7). O *EPOS Mote* é um *mote* para RSSF o qual possui todo o projeto de hardware aberto e executa o sistema EPOS (LISHA, 2010).

Para avaliar o desempenho dos adaptadores de código nativo desenvolvidos nesta aplicação, foi calculado o *overhead* de tempo de acesso aos métodos nativos. A Tabela 5 apresenta os resultados para a arquitetura AVR8 utilizando a FFI da KESO. Na Tabela 6 são apresentados os resultados para a arquitetura ARM7 utilizando a FFI da NanoVM. Para o sensor de temperatura foi medido o tempo do método *sample* que amostra a temperatura ambiente. Para a NIC mediu-se o tempo de envio de uma mensagem qualquer. Os tempos "Total" e "de dispositivo" entre as duas tabelas não são comparáveis, pois são dispositivos distintos, de plataformas distintas. O *overhead* percentual tente a ser maior para a FFI da NanoVM, pois na NanoVM existe o tempo gasto na interpretação de *bytecode* JAVA, o que não existe no caso da JVM KESO uma vez que o *bytecode* JAVA é traduzido para C em tempo de geração do sistema. Entretanto, nos valores da Tabela 6 não está sendo considerado este tempo de interpretação de *bytecode* JAVA e sim apenas o *overhead* gerado por operações de "push" e "pop" na pilha da NanoVM para obtenção dos argumentos dos métodos e retorno dos resultados.

O *overhead* de memória para os adaptadores de sensor de tem-

Tabela 7: *Overhead* de memória. Arquitetura AVR8, FFI da KESO

Seção	**Overhead (byte)**	**Footprint (byte)**	**Impacto (%)**
Temp::sample			
text	42	17654	0.24
data	0	207	0
bss	0	374	0
total	42	18235	0.23
NIC::send			
text	54	17654	0.31
data	0	207	0
bss	0	374	0
total	54	18235	0.3
Temp::sample+NIC::send			
text	96	17654	0.54
data	0	207	0
bss	0	374	0
total	96	18235	0.53

peratura e interface de rede é apresentado nas tabelas 7 e 8. O *footprint* indicado é da aplicação que executa no nodo *sensor*. O *footprint* obtido para a aplicação *sink*, não mostrado nas tabelas, é ainda menor pois considera apenas o adaptador de código nativo da NIC.

A aplicação desenvolvida para a NanoVM possui um *footprint* cerca de 2.36 vezes maior do que a equivalente desenvolvida para a KESO JVM. Isto se deve ao fato de que a NanoVM, como mencionado no Capítulo 3, carrega na imagem final do sistema, o interpretador de *bytecode* JAVA. No caso da JVM KESO não há necessidade de interpretador de *bytecode*, uma vez que o *bytecode* é traduzido para C no momento de geração do sistema. Observa-se também que o impacto dos adaptadores de código nativo é maior para a NanoVM do que para a KESO. Isto ocorre pois, como os mediadores de hardware utilizam técnicas de metaprogramação na sua implementação e ambos NanoVM e mediadores de hardware estão escritos em C++, os mediadores se dissolvem completamente nos adaptadores de código nativo. No caso da KESO isto não ocorre, pois o adaptador está escrito em C e o compilador C não entende as construções metaprogramadas de C++, impedindo que o mediador de hardware seja dissolvido no adaptador.

Tabela 8: *Overhead* de memória. Arquitetura ARM7, FFI da NanoVM

Seção	**Overhead (byte)**	**Footprint (byte)**	**Impacto (%)**
Temp::sample			
text	136	36784	0.37
data	0	289	0
bss	0	6052	0
total	136	43125	0.32
NIC::send			
text	1036	36784	2.82
data	0	289	0
bss	0	6052	0
total	1036	43125	2.4
Temp::sample+NIC::send			
text	1172	36784	3.19
data	0	289	0
bss	0	6052	0
total	1172	43125	2.72

Tabela 9: Síntese das avaliações realizadas.

Método	**Desempenho**	**Memória**	**Arquiteturas**	**FFIs**
UART:: put	Tabela 1	Tabela 2	IA32, AVR8	KESO
DMEC:: match	Tabela 3	Tabela 4	IA32	KESO, Lua
Temp:: sample	Tabelas 5 e 6	Tabelas 7 e 8	AVR8, ARM7	KESO, NanoVM
NIC:: send	Tabelas 5 e 6	Tabelas 7 e 8	AVR8, ARM7	KESO, NanoVM

5.3 DISCUSSÃO

A Tabela 9 sintetiza todos os dados obtidos nas avaliações dos três casos de estudo realizados. O mediador de hardware da UART foi avaliado em PC e no EPOS Mote. Os adaptadores de código nativo do DMEC foram avaliados em FFIs de duas MPLs distintas: JAVA e LUA. Os adaptadores do DMEC foram avaliados apenas na arquitetura IA32 pois, o DMEC, por possuir um alto grau de paralelismo, utiliza estruturas de dados replicadas e consequentemente consome aproximadamente 1 Megabyte de memória, valor acima do disponível na plataforma EPOS Mote (8KB versão AVR8 e 96KB versão ARM7). De forma similar, o adaptador para o sensor de temperatura só está disponível no EPOS Mote e não no PC onde o DMEC foi testado.

Além dos aspectos mensurados, a partir da análise dos casos de estudos, duas outras questões foram observadas: a geração de adaptadores de código nativo para abstrações de SO e portabilidade software/hardware.

Adaptadores de código nativo para abstrações de SO

Os programas *sensor* e *sink* (Figuras 49 e 50) demostram que, utilizando o método proposto, é possível desenvolver adaptadores de código nativo não apenas para mediadores de hardware como também para abstrações de SO. Nos programas *sensor* e *sink*, observa-se a utilização da abstração *PeriodicThread* que representa uma *thread* periódica, ou seja, uma *thread* que executa de tempos e tempos, de acordo com um período específico, o qual é definido no momento de sua criação.

Portabilidade software/hardware

Além de portabilidade entre plataformas de hardware distintas, existe portabilidade entre software e hardware. Portabilidade entre software e hardware significa que um mesmo adaptador de código nativo pode ser utilizado tanto em uma implementação de software quanto em uma implementação em hardware do componente sendo abstraído. Isto é possível com a utilização do conceito de *componentes híbridos* realizados pelo EPOS, onde um componente preserva a mesma interface tanto em sua implementação em software quanto em sua implementação em hardware (MARCONDES; FRöHLICH, 2009). O DMEC, por exemplo, atualmente é implementado por componentes de software, onde os módulos *Coordinator* e *Workers* são *threads* executando em núcleos distintos de um processador multinúcleo. Apesar disso, utilizando-se o conceito de componentes híbridos, o DMEC pode ser implementado por componentes de hardware preservando as mesmas interfaces disponíveis na versão em software. Neste caso, os adaptadores de código nativo desenvolvidos também permanecem os mesmos. Em um cenário de implementação em hardware *Coordinator* e *Worker* são IPs (do inglês *Intellectual Property* - IP) de um Multiprocessador em Chip (do inglês *Multiprocessor System-on-Chip* - MPSoC) e a comunicação entre eles é realizada por um sistema de interconexão em chip, como por exemplo os descritos por (JAVAID et al., 2010) e (POPOVICI; JERRAYA, 2009).

6 CONCLUSÕES

Esta dissertação investiga a questão de como realizar a interface entre dispositivos de hardware e MPLs de forma adequada para sistemas embarcados, ou seja, com baixo *overhead* de tempo e uso eficiente de memória. Outros requisitos considerados são a portabilidade dos adaptadores de código nativo entre plataformas de hardware distintas e o reuso desses adaptadores em diversas FFIs.

Para a realização da interface entre MPLs e dispositivos de hardware é proposto um método que especifica como os adaptadores de código nativo podem ser desenvolvidos. Por meio de mediadores de hardware é possível desenvolver adaptadores de código nativo com um baixo *overhead* de tempo e com consumo eficiente de recursos. Os mediadores de hardware garantem também a portabilidade dos adaptadores de código nativo entre diferentes plataformas de hardware. O reuso dos adaptadores de código nativo é atingido fatorando os mesmos em parte funcional (representada pelos mediadores de hardware) e parte não funcional (representada pela descrição da regras da FFI alvo). As partes não funcionais de um adaptador de código nativo são fatoradas como aspectos e a composição destas com os mediadores de hardware é resolvido como um problema de *weaving* de aspectos.

Aplicou-se o método proposto nas linguagens JAVA e LUA. No caso do JAVA as FFIs utilizadas foram a FFI do KESO e a FFI da NanoVM. Para a linguagem LUA utilizou-se a FFI padrão de LUA. Foi desenvolvido um adaptador de código nativo para cada mediador a ser utilizado nas aplicações. Este adaptador é construído utilizando-se a ferramenta EBG a qual possui como entrada uma descrição do mediador de hardware e a seleção da FFI alvo. A composição dos mediadores de hardware e a parte não funcional dos adaptadores de código nativo é realizada pelo módulo *Weaver* do EBG. A saída do EBG é o adaptador de código nativo customizado à FFI alvo e a contraparte em MPL do mediador de hardware. Os adaptadores, a aplicação e o ambiente de suporte à execução são compilados em conjunto para a geração da imagem final.

Compararam-se aplicações JAVA e LUA que acessam dispositivos de hardware utilizando os adaptadores de código nativo desenvolvidos utilizando-se a abordagem proposta com aplicações utilizando adaptadores de código nativo desenvolvidos de forma tradicional. As aplicações foram avaliadas em termos de desempenho total, *overhead* de tempo da FFI, consumo de memória total, *overhead* no consumo de

memória causado pela FFI, portabilidade entre diferentes plataformas de hardware e reuso entre diferentes FFIs.

Para a aplicação utilizando o mediador de hardware da UART o *overhead* de tempo obtido foi menor que 0.04% do tempo total de execução da aplicação quando usando um adaptador de código nativo desenvolvido utilizando-se o método proposto, o que corresponde a 38 vezes mais rápido do que se utilizando adaptadores de código nativo desenvolvidos utilizando-se a JNI da Sun. Utilizando-se a abordagem proposta nesta dissertação foram desenvolvidas aplicações para as arquiteturas IA32, AVR8 e ARM7. Visando avaliar a abordagem apresentada nesta dissertação em aplicações reais, foram desenvolvidos adaptadores de código nativo para um componente o qual realiza estimativa de movimento para codificação de vídeo H.264 e em uma aplicação distribuída que realiza monitoração de temperatura e se comunica por meio de rádio.

Considera-se que os objetivos deste trabalho foram atingidos pois, os resultados obtidos demonstram que aplicações MPL desenvolvidas utilizando os adaptadores de código nativo gerados de acordo com o método proposto, estão adequados para sistemas embarcados em termos de consumo de tempo, memória, portabilidade e reuso.

A principal contribuição deste trabalho é um método de como realizar a interface entre dispositivos de hardware e MPLs de forma adequada para sistemas embarcados. Até onde se pode identificar, de acordo com os trabalhos relacionados, nenhum método semelhante foi apresentado antes, as soluções existentes estavam apenas preocupadas em prover FFIs e implementações de MPLs para sistemas embarcados mas careciam de um padrão de abstração de dispositivos de hardware.

O método proposto pode ser utilizado na construção de bibliotecas de componentes de hardware para serem utilizados em aplicações MPL para sistemas embarcados. Os casos de estudo mostrados neste trabalho apresentam a aplicação do método proposto em JAVA e LUA, mas este pode ser aplicado, em princípio, em todas MPLs que não provêm acesso direto a memória ou o conceito de *inline assembly*, para controle direto a dispositivos de hardware.

Dentre as limitações deste trabalho estão a não avaliação do método proposto quanto a questões de tempo real e de consumo de energia. Avaliou-se o *overhead* de tempo dos adaptadores para cada um dos métodos nativos, mas não foram considerados dispositivos de hardware sensíveis a *jitter* e questões de previsibilidade temporal. De forma similar, não foram realizadas medições de energia para avaliar o impacto dos adaptadores de código nativo sobre os requisitos energéticos do

sistema.

Além de avaliar o método proposto sobre os aspectos de tempo real e energia, identifica-se como trabalho futuro a avaliação da correção dos adaptadores de código nativo gerados e a integração do método proposto com abordagens de MDE. A avaliação quanto a correção dos adaptadores de código nativo pode ser realizada utilizando ferramentas de análise de código fonte ou ferramentas de detecção dinâmica de erros, como as descritas na Seção 3.3. O método proposto nesta dissertação pode ser integrado com abordagens de MDE traduzindo os modelos de uma ferramenta MDE descritos, por exemplo, em UML para o metamodelo DERCS modificado, utilizado na ferramenta EBG. Desta forma, mediadores de hardware podem ser descritos, por exemplo, em UML e, a partir destes, a EBG pode gerar os adaptadores de código nativo necessários.

Resultados preliminares da aplicação do método proposto porém, ainda sem a utilização da ferramenta EBG, encontram-se publicados em (LUDWICH; FRöHLICH, 2011a) e em (LUDWICH; FRöHLICH, 2011b). As ideias preliminares de utilizar a abordagem proposta para prover as MPLs com abstrações de sistemas operacionais (além de componentes de hardware) são descritas em (LUDWICH; FRöHLICH, 2009).

REFERÊNCIAS

ARM. *Jazelle*. 2011. Disponível em: http://www.arm.com/products/processors/technologies/jazelle.php Acesso em 07 julho 2011.

ARNOLD, K.; GOSLING, J.; HOLMES, D. *The Java Programming Language*. 4th. ed. [S.l.]: Prentice Hall, 2005.

BAUER, H. *Entwurf eines OSEK Adaption Layers für das Betriebssystem EPOS*. Dissertação (Mestrado) — Friedrich-Alexander-Universität, Erlangen-Nurnberg, Deutschland, 2008.

BØGHOLM, T. et al. A predictable java profile: rationale and implementations. In: *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, NY, USA: ACM, 2009. (JTRES '09), p. 150–159. ISBN 978-1-60558-732-5. Disponível em: <http://doi.acm.org/10.1145/1620405.1620427>.

BØGHOLM, T. et al. Schedulability analysis for java finalizers. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, NY, USA: ACM, 2010. (JTRES '10), p. 1–7. ISBN 978-1-4503-0122-0. Disponível em: <http://doi.acm.org/10.1145/1850771.1850772>.

BOND, M. D.; MCKINLEY, K. S. Leak pruning. In: *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2009. (ASPLOS '09), p. 277–288. ISBN 978-1-60558-406-5. Disponível em: <http://doi.acm.org/10.1145/1508244.1508277>.

BROUWERS, N.; LANGENDOEN, K.; CORKE, P. Darjeeling, a feature-rich vm for the resource poor. In: *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. New York, NY, USA: ACM, 2009. p. 169–182. ISBN 978-1-60558-519-2.

CTYPESLIB. *ctypeslib - useful additions to the ctypes FFI library*. 2011. Disponível em: http://pypi.python.org/pypi/ctypeslib/. Acesso em 07 julho 2011.

DIJKSTRA, E. W. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal*

Perspective, Springer-Verlag, p. 60–66, 1982. Disponível em: <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>.

ESMAEILZADEH, H. et al. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2011. (ASPLOS '11), p. 319–332. ISBN 978-1-4503-0266-1. Disponível em: <http://doi.acm.org/10.1145/1950365.1950402>.

FRENZ, S. *SJC: Small Java Compiler*. 2011. Disponível em: http://www.fam-frenz.de/stefan/compiler.html Acesso em 07 julho 2011.

FRöHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. (GMD Research Series, 17).

FRöHLICH, A. A.; Schröder-Preikschat, W. Scenario adapters: Efficiently adapting components. In: . Orlando, USA: [s.n.], 2000.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. [S.l.]: Addison-Wesley Professional, 1995.

GINGA. *Ginga*. 2011. Disponível em: http://www.ginga.org.br/. Acesso em 07 julho 2011.

GOLM, M. et al. The jx operating system. In: *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002. p. 45–58. ISBN 1-880446-00-6.

GOOGLE. *Android*. 2008. Disponível em: http://www.android.com/. Acesso em 07 julho 2011.

GROUP, O. *OSEK/VDX Operating System Version 2.2.3*. fev. 2005.

H.264, R. *Rede H.264 SBTVD*. 2009. Disponível em: http://www.lapsi.eletro.ufrgs.br/h264/wiki/tiki-index.php. Acesso em 07 julho 2011.

HARBAUM, T. *The NanoVM - Java for the AVR*. 2005. Disponível em: http://www.harbaum.org/till/nanovm/index.shtml. Acesso em 07 julho 2011.

HIRZEL, M.; GRIMM, R. Jeannie: granting java native interface developers their wishes. In: *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. New York, NY, USA: ACM, 2007. p. 19–38. ISBN 978-1-59593-786-5.

IERUSALIMSCHY, R. *Programming in Lua*. 2nd. ed. [S.l.]: Lua.org, 2006.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. The implementation of lua 5.0. *Journal of Universal Computer Science*, v. 11, n. 7, p. 1159–1176, jul 2005.

JAVAID, H. et al. Optimal synthesis of latency and throughput constrained pipelined mpsocs targeting streaming applications. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2010. (CODES/ISSS '10), p. 75–84. ISBN 978-1-60558-905-3. Disponível em: <http://doi.acm.org/10.1145/1878961.1878978>.

JAVA.NET. *PhoneME*. 2006. Disponível em: http://java.net/projects/phoneme/. Acesso em 07 julho 2011.

KICZALES, G. et al. Aspect-oriented programming. In: *ECOOP*. [S.l.]: SpringerVerlag, 1997.

KONDOH, G.; ONODERA, T. Finding bugs in java native interface programs. In: *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2008. p. 109–118. ISBN 978-1-60558-050-0.

KORSHOLM, S.; JEAN, P. The java legacy interface. In: *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. New York, NY, USA: ACM, 2007. p. 187–195. ISBN 978-59593-813-8.

LEE, B. et al. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 45, p. 36–49, June 2010. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1809028.1806601>.

LI, X.; LI, E.; CHEN, Y.-K. Fast multi-frame motion estimation algorithm with adaptive search strategies in h.264. In: . [S.l.: s.n.], 2004. v. 3, p. iii – 369–72 vol.3. ISSN 1520-6149.

LIANG, S. *The Java Native Interface - Programmer's Guide and Specification*. [S.l.]: Addison-Wesley, 1999.

LISHA. *EPOS Mote*. 2010. Disponível em: http://epos.lisha.ufsc.br/EPOSMote Acesso em 07 julho 2011.

LUA.ORG. *Lua: about*. 2011. Disponível em: http://www.lua.org/about.html. Acesso em 07 julho 2011.

LUDWICH, M. K.; FRöHLICH, A. A. Interfacing operating systems components with embedded java applications. In: *International Information and Telecommunication Technologies Symposium*. Florianópolis, Brazil: [s.n.], 2009. p. 203–206. ISBN 978-85-89264-10-5.

LUDWICH, M. K.; FRöHLICH, A. A. Abstracting hardware devices to embedded java applications. In: *IADIS Applied Computing 2011*. Rio de Janeiro, Brazil: [s.n.], 2011. p. 371–378. ISBN 978-989-8533-06-7.

LUDWICH, M. K.; FRöHLICH, A. A. Abstraindo dispositivos de hardware para aplicações java embarcadas. In: *Brazilian Symposium on Computing System Engineering*. Florianópolis, Brazil: [s.n.], 2011. p. 348–359.

LUDWICH, M. K.; FRöHLICH, A. A. Optimizing motion estimation for h.264 encoding. In: *XVII Brazilian Symposium on Multimedia and the Web*. Florianópolis, Brazil: [s.n.], 2011. p. 198–204.

MACHADO, A. de M.; FRöHLICH, A. A. A lua virtual machine for resource-constrained embedded systems. In: *IADIS International Conference Applied Computing 2010*. Timisoara, Romania: [s.n.], 2010. p. 175–182. ISBN 978-972-8939-30-4.

MARCONDES, H.; FRöHLICH, A. A. A Hybrid Hardware and Software Component Architecture for Embedded System Design. In: *International Embedded System Symposium*. Langenargen, Germany: [s.n.], 2009. p. 259–270. ISBN 978-3-642-04283-6.

MICROSYSTEMS, I. S. *J2ME Building Blocks for Mobile Devices*: White paper on kvm and the connected, limited device configuration (cldc). [S.l.]: Sun Microsystems, Inc., 2000.

MICROSYSTEMS, I. S. *K Native Interface (KNI)*. [S.l.]: Sun Microsystems, Inc., 2002.

ORACLE. *Java Communications API*. 2011. Disponível em: http://www.oracle.com/technetwork/java/index-jsp-141752.html. Acesso em 07 julho 2011.

ORACLE. *The Java HotSpot Performance Engine Architecture*. 2011. Disponível em: http://www.oracle.com/technetwork/java/whitepaper-135217.html. Acesso em 07 julho 2011.

ORACLE. *Java ME Landing Page*. 2011. Disponível em: http://www.oracle.com/technetwork/java/javame/index.html. Acesso em 07 julho 2011.

PHIPPS, G. Comparing observed bug and productivity rates for java and c++. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 29, n. 4, p. 345–358, abr. 1999. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/(SICI)1097-024X(19990410)29:4¡345::AID-SPE238¿3.0.CO;2-C>.

PIZLO, F. et al. High-level programming of embedded hard real-time devices. In: *Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010. (EuroSys '10), p. 69–82. ISBN 978-1-60558-577-2. Disponível em: <http://doi.acm.org/10.1145/1755913.1755922>.

POLPETA, F. V. *Uma Estratégia para a Geração de Sistemas Embutidos baseada na Metodologia Projeto de Sistemas Orientados à Aplicação*. Dissertação (Mestrado) — Federal University of Santa Catarina, Florianópolis, 2006.

POLPETA, F. V.; FRöHLICH, A. A. Hardware mediators: a portability artifact for component-based systems. In: *In Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of LNCS, Aizu,Japan*. [S.l.]: Springer, 2004. p. 271–280.

POPOVICI, K.; JERRAYA, A. Flexible and abstract communication and interconnect modeling for mpsoc. In: *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 2009. (ASP-DAC '09), p. 143–148. ISBN 978-1-4244-2748-2. Disponível em: <http://portal.acm.org/citation.cfm?id=1509633.1509681>.

PORTAL, O. V. *OSEK VDX Portal*. 2008. Disponível em: <http://www.osekvdx.org/>. Acesso em: 20 aug. 2008.

PROJECT eLua. *eLua - eluaproject*. 2011. Disponível em: http://www.eluaproject.net/. Acesso em 07 julho 2011.

PUFFITSCH, W.; SCHOEBERL, M. picojava-ii in an fpga. In: *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. New York, NY, USA: ACM, 2007. p. 213–221. ISBN 978-59593-813-8.

RAVITCH, T. et al. Automatic generation of library bindings using static analysis. In: *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009. (PLDI '09), p. 352–362. ISBN 978-1-60558-392-1. Disponível em: <http://doi.acm.org/10.1145/1542476.1542516>.

RXTX. *RXTX: serial and parallel I/O libraries supporting Sun's CommAPI*. 2011. Disponível em: http://rxtx.qbang.org/wiki/index.php/Main_Page. Acesso em 07 julho 2011.

SAMPSON, A. et al. Enerj: approximate data types for safe and general low-power computation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 46, p. 164–174, June 2011. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1993316.1993518>.

SCHOEBERL, M. A java processor architecture for embedded real-time systems. *J. Syst. Archit.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 1-2, p. 265–286, 2008. ISSN 1383-7621.

STILKERICH, M. et al. An OSEK/VDX API for Java. In: ACM (Ed.). *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*. New York: [s.n.], 2006. p. 13–17. ISBN 1-59593-577-0.

SWIG. *Simplified Wrapper and Interface Generator*. 2011. Disponível em: http://www.swig.org/. Acesso em 07 julho 2011.

TAN, G.; MORRISETT, G. Ilea: inter-language analysis across java and c. In: *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. New York, NY, USA: ACM, 2007. p. 39–56. ISBN 978-1-59593-786-5.

TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

THOMM, I. et al. Keso: an open-source multi-jvm for deeply embedded systems. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, NY, USA: ACM, 2010. (JTRES '10), p. 109–119. ISBN 978-1-4503-0122-0. Disponível em: <http://doi.acm.org/10.1145/1850771.1850788>.

VELASCO, J. M.; ATIENZA, D.; OLCOZ, K. Exploration of memory hierarchy configurations for efficient garbage collection on high-performance embedded systems. In: *Proceedings of the 19th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2009. (GLSVLSI '09), p. 3–8. ISBN 978-1-60558-522-2. Disponível em: <http://doi.acm.org/10.1145/1531542.1531549>.

WAWERSICH, C.; STILKERICH, M.; SCHRÖDER-PREIKSCHAT, W. An OSEK/VDX-based Multi-JVM for Automotive Appliances. In: BOSTON, S. (Ed.). *Embedded System Design: Topics, Techniques and Trends*. Boston: [s.n.], 2007. (IFIP International Federation for Information Processing), p. 85–96. ISBN 978-0-387-72257-3. ISSN 1571-5736.

WEHRMEISTER, M. A. *An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems*. Tese (Ph.D. Thesis) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação., Porto Alegre, RS, Brazil, 2009.

WIEGAND, T. et al. Overview of the h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, v. 13, n. 7, p. 560–576, August 2003. Disponível em: <http://dx.doi.org/10.1109/TCSVT.2003.815165>.

ZERZELIDIS, A.; WELLINGS, A. A framework for flexible scheduling in the rtsj. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 10, p. 3:1–3:44, August 2010. ISSN 1539-9087. Disponível em: <http://doi.acm.org/10.1145/1814539.1814542>.