2011 Brazilian Symposium on Computing System Engineering

Abstraindo dispositivos de hardware para aplicações Java embarcadas (Interfacing hardware devices to embedded Java)

Mateus Krepsky Ludwich, Antônio Augusto Fröhlich
Laboratory for Software e Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 880400900 - Florianópolis - SC - Brasil
Email: {mateus,guto}@lisha.ufsc.br

***Abstract*—Access to hardware devices is an important requirement to be fulfilled by Java implementations targeting embedded systems because the interaction between the embedded system and the environment where it is inserted on is performed by these devices. In this paper we introduce a method for abstracting hardware devices to embedded Java applications. We have evaluated our method in terms of performance, memory footprint, and portability. The applicability of our method was tested for abstracting simple hardware devices for serial communication and for abstracting more complex components such as a motion estimator for H.264 video coding.**

***Keywords*-Java, Embedded Systems, FFI**

I. INTRODUÇÃO

As chamadas linguagens de altíssimo nível (VHLLs - *Very High Level Languages*) , da qual o Java é um exemplo, facilitam o desenvolvimento de sistemas embarcados pois elas provêm funcionalidades como orientação a objetos, gerenciamento automático de memória e proteção de memória. Além dos requisitos de tempo e de consumo de memória impostos por sistemas embarcados (SEs), outro importante requisito que deve ser atendido por implementações Java focadas em SEs é prover aos desenvolvedores um meio de controlar dispositivos de hardware. Isto é necessário uma vez que as aplicações para sistemas embarcados executam próximas do hardware, no sentido de que elas utilizam dispositivos de hardware como sensores e atuadores para interagir com o ambiente, transmissores e receptores para comunicação e temporizadores para operações em tempo real.

Interface de Função Estrangeira (FFI - *Foreign Function Interface*) é o mecanismo utilizado por plataformas Java para acessar dispositivos de hardware e memória. De fato, diversos pacotes Java como *java.io*, *java.net* e *java.awt* são implementados utilizando funcionalidades de FFI [1]. Entretanto as principais FFIs Java possuem limitações para lidar com sistemas embarcados. Elas são demasiadamente onerosas ou possuem limitações de projeto. Além disso, as FFIs por si só não auxiliam na abstração de dispositivos de hardware para VHLLs, elas apenas provêm um meio de utilizar código escrito em outras linguagens, como C e C++, capazes de acessar diretamente dispositivos de hardware.

Neste artigo apresentamos um método para realizar a interface entre dispositivos de hardware e aplicações Java embarcadas. Como forma de superar as limitações das principais FFIs Java para sistemas embarcados, utilizamos a FFI da máquina virtual Java (JVM - *Java Virtual Machine*) KESO [2]. A JVM KESO foca em sistemas embarcados e traduz programas em bytecode Java para C antes da execução do mesmos. A FFI da KESO também utiliza uma abordagem estática, gerando o código C especificado para a implementação de métodos nativos. Como forma de orientar e facilitar a realização de interface entre dispositivos de hardware e Java utilizamos o conceito de *mediadores de hardware*, propostos pela metologia de Projeto de Sistemas Embarcados Orientada pela Aplicação (ADESD - *Application-Driven Embedded System Design*) [3]. Os mediadores de hardware são semelhantes a *drivers* UNIX, abstraindo dispositivos de hardware para serem utilizados pelas demais abstrações de sistema (e.g. *threads*). O EPOS (*Embedded Parallel Operating System*), é um caso de aplicação da ADESD no domínio de sistemas operacionais, e implementa o conceito de mediadores de hardware [3].

A principal contribuição deste artigo é um método para suportar o desenvolvimento de bibliotecas de classes cujos elementos são abstrações de dispositivos de hardware. Demonstramos o uso do método proposto no desenvolvimento de adaptadores de código nativo (*bindings* FFI) de dispositivos simples para comunicação serial e de componentes mais complexos, como um componente que realiza estimativa de movimento em codificação de vídeo H.264.

As próximas seções deste artigo estão organizadas da seguinte maneira: a Seção II apresenta trabalhos relacionados envolvendo interface entre hardware e Java e abordagens de geração de adaptadores. Apresentamos nossa abordagem de interface entre dispositivos de hardware e aplicações Java na Seção III. Conceitos do EPOS e da KESO são apresentados quando necessário para suportar o entendimento da nossa proposta. Na Seção IV avaliamos nossa abordagem em termos de desempenho, consumo de memória e portabilidade. A Seção V apresenta um componente de estimativa de movimento para codificação H.264 como um exemplo de

978-0-7695-4641-4/11 $26.00 © 2011 IEEE
DOI 10.1109/SBESC.2011.32

IEEE computer society

utilização do nosso método para o desenvolvimento de uma aplicação real. Nossas considerações finais são apresentadas na Seção VI.

II. Trabalhos relacionados

A linguagem de programação Java é desprovida do conceito de *ponteiro*, presente em linguagens como C e C++. O endereço das *variáveis de referência*, utilizadas para acessar objetos Java, é conhecido apenas pela JVM, a qual trata de todos os acessos à memória. Como a maioria dos dispositivos de hardware são mapeados em endereços de memória, acessá-los diretamente é um problema para a linguagem Java. FFI é a abordagem utilizada por Java para superar esta limitação uma vez que ela permite ao Java utilizar construções, como ponteiros C/C++, para acessar diretamente dispositivos de hardware. FFIs também tem sido utilizadas por plataformas Java na reutilização de código escrito em outras linguagens de programação como C e C++ e para embarcar JVMs em aplicações nativas permitindo as mesmas acessar funcionalidades Java [1].

Java Native Interface (JNI) é a principal FFI Java, a qual é utilizada na plataforma *Java Standard Edition* [1]. Na JNI, 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 (métodos que possuem a palavra reservada *native* em suas assinaturas). Usualmente a implementação dos métodos nativos é armazenada em uma biblioteca ligada dinamicamente. 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 eles são evitados em sistemas embarcados.

A plataforma *Java Micro Edition* (JME) utiliza uma FFI "leve", chamada de *K Native Interface* (KNI) [4]. A KNI não carrega métodos nativos dinamicamente na JVM, evitando o sobrecusto 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 Aplicações (API - Application Programming Interface) em nível Java para invocar outros métodos nativos. Como consequência, é difícil de criar novos controladores de dispositivos de hardware utilizando-se a KNI.

A FFI da KESO, utilizada neste trabalho, foca em sistemas embarcados. Assim como a KNI, a FFI da KESO não realiza 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.

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. 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 [5],[6]. Ravitch 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 Ravitch 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 [7]. Outras soluções, como a linguagem *Jeannie*, misturam código C e Java em um único programa a partir do qual geram adaptadores JNI automaticamente [8]. A FFI da KESO, utilizada neste trabalho, provê uma API baseada em aspectos para ajudar na criação de adaptadores. É possível especificar quais pontos do programa Java serão afetados pela criação dos adaptadores, assim como qual código deve ser gerado para cada ponto do programa Java a ser afetado.

III. Interface entre dispositivos de hardware e linguagens de altíssimo nível

Nossa proposta para realizar a interface entre linguagens de programação de altíssimo nível e dispositivos de hardware é baseada em exportar estes dispositivos para a API da linguagem. Os dispositivos de hardware a serem exportados possuem uma interface bem definida, utilizando o conceito de mediadores de hardware da ADESD. Utilizamos a FFI da JVM KESO de forma a exportar mediadores EPOS para Java. Esta Seção explica como abstraímos componentes de hardware para serem utilizados por aplicações Java embarcadas. A utilização da abordagem é exemplificada abstraindo-se para Java um dispositivo de hardware transmissor-receptor assíncrono universal (UART - *Universal Asynchronous Receiver Transmitter*).

O EPOS utiliza o conceito de mediadores de hardware para abstrair especificidades de dispositivos de hardware distintos. Os mediadores de hardware sustentam um contrato de interface entre as abstrações de sistema (e.g. *threads*) e o hardware, permitindo à estas abstrações independência de plataforma [9]. A geração da implementação do mediador para uma plataforma específica é executada em tempo de compilação. Utilizando-se técnicas de meta programação e

inlining de funções é possível dissolver os mediadores entre as abstrações que os utilizam, o que evita sobrecusto de tempo no uso de mediadores.

A KESO provê uma FFI para realizar a interface entre código Java e código C ou C++. A FFI da KESO utiliza uma abordagem estática assim como a KNI da Sun, realizando a interface entre Java e código nativo em tempo de compilação.

O projeto da FFI da KESO adota alguns conceitos de Programação Orientada a Aspectos (AOP - Aspect-Oriented Programming). 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 1 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.

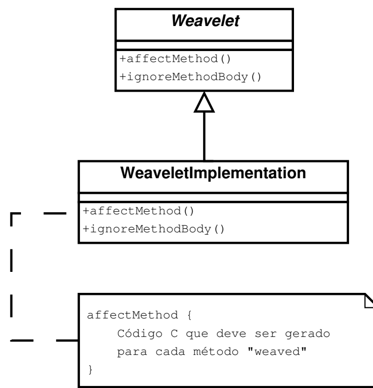

Figura 1. Utilizando a FFI KESO

Utilizamos a FFI do KESO para criar um adaptador para cada mediador EPOS que deve ser acessado pelo Java, provendo Java com componentes de hardware. A abordagem utilizada é mostrada na Figura 2. A classe Java, a qual representa a contraparte Java do mediador de hardware que está sendo abstraído, especifica assinaturas de métodos mas não os implementa, uma vez que tais métodos representam métodos nativos. Então, uma classe *weavelet* da FFI do KESO é utilizada para especificar a implementação de cada método nativo. Mais especificamente, a classe *weavelet* especifica quais métodos da classe Java deseja-se interceptar (o equivalente aos *pointcuts* de uma linguagem de AOP) e quais os respectivos códigos que devem ser gerados (o equivalente aos *advices* em AOP). Durante a tradução de bytecode Java em C, o compilador da KESO entrelaça as assinaturas dos métodos nativos da classe Java com os *advices* especificados pelo *weavelet*, gerando o código dos adaptadores que realizam a interface entre a classe Java e o mediador de hardware EPOS.

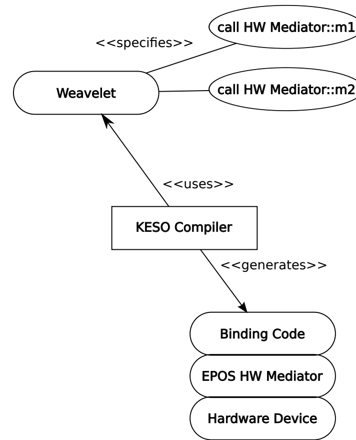

Figura 2. Interface Hardware/Java

A implementação de cada método nativo, especificada no *weavelet*, é basicamente uma chamada para cada método do mediador EPOS que está sendo exportado ao Java. Na implementação do método *affectMethod* é possível especificar um padrão o qual representa um elemento do programa Java (uma assinatura de método, por exemplo), assim como qual código deverá ser gerado quando o compilador KESO reconhece tal padrão. O trecho de código da Figura 3 mostra a especificação de um adaptador de código nativo a ser gerado para um método virtual chamado de *m1*. Este método possui um único parâmetro do tipo carácter e não possui retorno. O *eposHWMediator* é um campo da classe adaptadora o qual aponta para o objeto mediador de hardware do EPOS. É possível adicionar campos nas classes adaptadoras implementando-se o método *addFields* da classe abstrata *Weavelet*.

Quando cria-se um objeto Java (chamando-se *new*) que representa uma abstração de hardware, o objeto adaptador de código nativo, pode alocar objetos EPOS como por exemplo

```
public boolean affectMethod(IMClass clazz,
  IMMethod method,
  Coder coder)
throws CompileException
{
  // ...
  if (method.termed("m1(C)V")) {
    coder.addln("obj0->eposHWMediator->m1(c1);");
    return true;
  }
  // ...
}
```

Figura 3. Especificando os adaptadores de código nativo

```java
package test;
import keso.core.Task;

public class UART_Test extends Task {
  public void launch() {
    UART serial;
    serial = new UART(19200, 8, 0, 1, 0);
    for(int i = 0; i < 10000; i++) {
      serial.put('M');
    }
  }
}
```

Figura 4. Exemplo UART

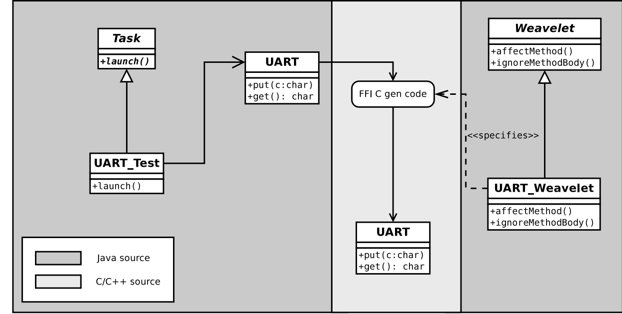

Figura 5. Arquitetura do programa UART

o *eposHWMediator* do programa da Figura 3. O objeto Java será desalocado automaticamente pelo coletor de lixo da KESO JVM e o objeto EPOS, segundo nossa abordagem, é desalocado chamando-se o método *finalizer* da classe Java. Como o coletor de lixo da KESO JVM utiliza algoritmos determinísticos é garantido que os *finalizers* de Java são sempre chamados.

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 estáticas executadas pelo compilador KESO, a FFI da KESO ainda apresenta algumas vantagens interessantes, as quais nos motivaram a utilizá-la. 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.

Escrevemos uma pequena aplicação utilizando o mediador de hardware UART para ilustrar nossa proposta de abstração de dispositivos de hardware para Java embarcado. A aplicação, mostrada pelo programa da Figura 4, utiliza a UART para escrever caracteres em um dispositivo serial.

A classe Java *UART* é a contraparte Java da classe UART do mediador de hardware EPOS e possui apenas métodos nativos sem quaisquer implementação. Então, o *UART_Weavelet* é utilizado para especificar a implementação de cada método da *UART*. A abordagem é a mesma apresentada pelo programa da Figura 3. A Figura 5, por sua vez, mostra a arquitetura geral do programa exemplo UART, mostrando as principais classes envolvidas e distinguindo a parte de código Java da parte de código C/C++. 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*.

IV. AVALIAÇÃO

Avaliamos nossa proposta de interface entre dispositivos de hardware e Java, descrita na Seção III, em termos de desempenho, consumo de memória e portabilidade. Utilizamos em nossos experimentos o exemplo da UART apresentado na Seção III e um componente para estimativa de movimento em codificação de vídeo H.264. Esta Seção descreve os resultados obtidos para o mediador UART. A Seção V descreve em detalhes o componente de estimativa de movimento assim como os resultados obtidos na análise do mesmo.

Em todos os experimentos, o código C/C++ gerado pelo compilador do KESO e pela FFI da KESO foi compilado para código nativo utilizando o GCC (gcc e g++). A aplicação UART foi avaliada nas arquiteturas IA32 e PowerPC 32 (PPC32) e a aplicação de estimativa de movimento foi avaliada na arquitetura IA32. O código fonte do EPOS e da KESO JVM estão disponíveis para download, respectivamente em [10] e em [11].

A. Desempenho

Um adaptador que realize interface entre linguagens de altíssimo nível 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 da Seção III, medimos o tempo de resposta do dispositivo de hardware em análise acessando-o diretamente (e.g. por uma aplicação C++) e acessando-o por nossos adaptadores de código nativo. O sobrecusto de tempo gerado por uma FFI pode ser descrito pela equação 1. Definimos *TempoDispositivo* como o tempo de resposta original o qual é composto pelo tempo do mediador EPOS mais o tempo do dispositivo físico. O *TempoTotal* adiciona ao *TempoDispositivo* o tempo de resposta do método nativo, incluindo a chamada para o método e o retorno do método.

$$SobrecustoFFI(\%) = \left(1 - \frac{TempoDispositivo}{TempoTotal}\right) \times 100 \tag{1}$$

Medimos o *TempoTotal* e o *TempoDispositivo* do método *put* da UART. O método foi chamado 10 mil vezes em uma execução de aplicação e a aplicação foi executada 30 vezes. Utilizamos o *time stamp clock* do EPOS para computar o tempo. O sobrecusto obtido, por meio da equação 1,

	Total(μs)	UART(μs)	Sobrecusto FFI (%)
Proposta	517.74	517.54	0.04
JSE	8364683.74	8238695.07	1.5

Tabela I
SOBRECUSTO DE TEMPO GERADO PELA FFI

	IA32(byte)	PPC32(byte)
text	28645	30504
data	1180	1198
bss	1264	840
total	31089	32542

Tabela II
CONSUMO DE MEMÓRIA TOTAL

corresponde a menos que 0.04% do tempo total de execução do método.

Como forma de estimar a relevância do valor do sobrecusto, reproduzimos o experimento da UART utilizando a plataforma *Java Standard Edition* a qual utiliza a JNI como FFI. Utilizamos a biblioteca *RXTX* a qual implementa a *Java Communications API* , utilizada para comunicação com dispositivos seriais [12]. A função *gettimeoftheday* foi utilizada para computar o *TempoDispositivo* e o método Java *System.nanoTime* foi utilizado para computar o *TempoTotal*. A Tabela I mostra os valores obtidos, assim como os valores obtidos utilizando-se a nossa proposta.

A aplicação baseada em JNI apresenta um sobrecusto de 1.5% o qual é aproximadamente 38 vezes maior do que o sobrecusto obtido utilizando-se nossa abordagem. Considerando uma aplicação que precisa enviar um byte a cada 420μs e considerando que a UART leva 417μs para enviar um byte (*baud rate* de 19200), um sobrecusto de 1.5% (6.25μs) comprometeria na transmissão dos dados, ocasionando perda de *deadlines* em uma aplicação de tempo real.

B. *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. Como forma de estimar o sobrecusto de memória gerado pela nossa abordagem, medimos o consumo de memória (*footprint*) da imagem binária contendo o sistema como um todo (aplicação, JVM e ambiente de suporte a execução).

A Tabela II mostra os valores obtidos para o exemplo UART, o qual foi compilado para as arquiteturas *IA32* e *Power PC32*. O tamanho do sistema como um todo incluindo a aplicação, a JVM KESO e o EPOS possui menos de 33KB em ambas as arquiteturas, um valor adequado para diversas plataformas embarcadas.

C. *Portabilidade*

Nossos adaptadores de código nativo suportam dois tipos de portabilidade: portabilidade de plataforma e portabilidade entre software e hardware.

Uma vez que um adaptador de código nativo desenvolvido utilizando nossa abordagem utiliza o conceito de mediadores de hardware EPOS e este provê uma interface independente de máquina, nossos adaptadores podem existir para todas as arquiteturas e plataformas suportadas pelo EPOS.

Por portabilidade entre software e hardware queremos dizer 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 graças ao conceito de *componentes híbridos* realizados pelo EPOS, aonde um componente preserva a mesma interface tanto em sua implementação em software quanto em sua implementação em hardware [13].

V. APLICAÇÃO REAL

Como forma de avaliar nossa proposta em uma aplicação real, desenvolvemos adaptadores de código nativo para um componente o qual realiza estimativa de movimento (ME - *Motion Estimation*) em codificação de vídeo H.264. ME é utilizada para explorar a similaridade entre imagens vizinhas em uma sequencia de vídeo, permitindo que elas sejam codificadas diferencialmente, aumentando a taxa de compressão do *bitstream* gerado. ME é um estágio significante da codificação H.264, pois ele consome aproximadamente 90% do tempo total do processo de codificação [14].

O componente de calculo de ME, utiliza uma estratégia de particionamento de dados, onde a estimativa de movimento de cada partição da imagem é realizada em paralelo, em unidades funcionais específicas (e.g. em núcleos de um processador multinúcleo). Por esta razão o componente é chamado de *Distributed Motion Estimation Component* - DMEC (Componente de Estimativa de Movimento Distribuída).

Entretanto toda a complexidade na implementação do DMEC, cujo objetivo é aumentar o desempenho da ME, é ocultada da aplicação Java (e.g. codificador H.264), a 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 Java utilizando a estratégia apresentada na Seção III, aonde é desenvolvido um adaptador de código nativo para cada objeto sendo abstraído.

Atualmente o DMEC é implementado por componentes de software, onde os módulos que calculam a ME em cada partição da imagem executam em threads e cada uma das threads executa em um núcleo distinto de um processador multinúcleo. Apesar disso, o DMEC pode ser implementado por componentes de hardware preservando as mesmas interfaces disponíveis na versão em software. Isso pode ser obtido por meio o uso do conceito de *componentes*

```
public class DmecApp extends Task {
  public void launch() {
    int width = 1920; int height = 1088; int mrp = 1;
    PME estimator = new PME(width, height, mrp);

    Picture cp = TS.createPicture(width, height);
    Picture[] list0 = new Picture[2];
    for (int i = 0; i < list0.length; i++) {
        list0[i] = TS.createPicture(width, height);
    }

    PMC mvsc = estimator.match(cp, list0);

    TS.testPMC(mvsc, width, height, cp, list0);
  }
}
```

Figura 6. Aplicação Java DMEC

híbridos do EPOS. Neste caso, nossos adaptadores de código nativo também permanecem os mesmos. Em um cenário de implementação em hardware os módulos que calculam a ME são IPs de um multiprocessador em chip e a comunicação entre eles é realizada por um sistema de interconexão em chip, como por exemplo os descritos por [15] e [16].

Escrevemos uma aplicação 100% Java para utilizar nosso componente DMEC. 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 programa da Figura 6 mostra o principal método da aplicação. O DMEC é utilizado como um objeto Java usual.

VI. Conclusões

Neste artigo, apresentou-se um meio de realizar a interface entre componentes de hardware e aplicações Java para sistemas embarcados. Isto foi obtido utilizando-se a FFI da JVM KESO e o EPOS.

Avaliamos nossa abordagem em termos de desempenho, consumo de memória e portabilidade. Para a aplicação utilizando o mediador de hardware da UART o sobrecusto de tempo obtido foi menos de 0.04 % do tempo total de execução da aplicação e nossa solução é 38 vezes mais rápido do que a JNI da Sun. O consumo de memória para tal aplicação foi de 33KB, incluindo todo o suporte de ambiente de execução, o qual é adequado para diversos sistemas embarcados. Utilizando o EPOS obtivemos portabilidade para várias plataformas e utilizando o conceito de componentes híbridos podemos utilizar os mesmos adaptadores de código nativo tanto para componentes implementados em hardware como implementados em software.

Visando avaliar nossa abordagem em uma aplicação real, escrevemos adaptadores de código nativo para um componente o qual realizada estimativa de movimento para codificação de vídeo H.264.

Referências

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

[2] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat, "Keso: an open-source multi-jvm for deeply embedded systems," in *JTRES '10*. New York, NY, USA: ACM, 2010, pp. 109–119.

[3] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[4] I. Sun Microsystems, *K Native Interface (KNI)*. Sun Microsystems, Inc., 2002.

[5] SWIG, "Simplified wrapper and interface generator," 2011, disponível em: http://www.swig.org/. Acesso em 07 julho 2011.

[6] CTYPESLIB, "ctypeslib - useful additions to the ctypes ffi library," 2011, disponível em: http://pypi.python.org/pypi/ctypeslib/. Acesso em 07 julho 2011.

[7] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit, "Automatic generation of library bindings using static analysis," in *PLDI '09*. New York, NY, USA: ACM, 2009, pp. 352–362.

[8] M. Hirzel and R. Grimm, "Jeannie: granting java native interface developers their wishes," in *OOPSLA '07*. New York, NY, USA: ACM, 2007, pp. 19–38.

[9] F. V. Polpeta and A. A. Fröhlich, "Hardware mediators: a portability artifact for component-based systems," in *EUC'04*. Springer, 2004, pp. 271–280.

[10] EPOS, "Embedded parallel operating system," 2001, disponível em: http://epos.lisha.ufsc.br Acesso em 07 julho 2011.

[11] KESO, "Keso," 2008, disponível em: http://www4.informatik.uni-erlangen.de/Research/KESO. Acesso em 07 julho 2011.

[12] 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.

[13] H. Marcondes and A. A. Fröhlich, "A Hybrid Hardware and Software Component Architecture for Embedded System Design," in *International Embedded System Symposium*, Langenargen, Germany, Sep. 2009, pp. 259–270.

[14] X. Li, E. Li, and Y.-K. Chen, "Fast multi-frame motion estimation algorithm with adaptive search strategies in h.264," vol. 3, may. 2004, pp. iii – 369–72 vol.3.

[15] H. Javaid, X. He, A. Ignjatovic, and S. Parameswaran, "Optimal synthesis of latency and throughput constrained pipelined mpsocs targeting streaming applications," in *CODES/ISSS '10*. New York, NY, USA: ACM, 2010, pp. 75–84.

[16] K. Popovici and A. Jerraya, "Flexible and abstract communication and interconnect modeling for mpsoc," in *ASP-DAC '09*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 143–148.