UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

André Quintino Kuhnen

UMA MÁQUINA VIRTUAL JAVA MULTI-THREAD PARA SISTEMAS EMBARCADOS

Florianópolis

2012

André Quintino Kuhnen

UMA MÁQUINA VIRTUAL JAVA MULTI-THREAD PARA SISTEMAS EMBARCADOS

Tese submetida ao Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciência da Computação.
Orientador: Prof. Dr. Antônio Augusto Medeiros Frohlich
Coorientador: Mateus Krepsky Ludwich

Florianópolis

2012

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Monografia sob o título “Uma Máquina Virtual Java Multi-Thread para Sistemas Embarcados”, defendida em 5 de setembro de 2012 por André Quintino Kuhnen, como parte dos requisitos à obtenção do grau de Bacharel em Ciência da Computação, e aprovada pela seguinte banca examinadora:

Prof. Dr. Antônio Augusto M. Fröhlich
Orientador

Me. Mateus Krepsky Ludwich

Me. Arliones Stevert Hoeller Junior

Prof. Dr. Rafael Luiz Cancian

Dedico este trabalho à minha família pelo suporte que me foi fornecido e a minha mulher pela compreensão e ajuda.

AGRADECIMENTOS

Agradeço aos meus orientadores, os professores e colegas do curso por engrandecer minhas faculdades mentais. E aos músicos por deixar as horas dedicadas ao estudo mais agradável.

Dentro da boca é escuro

Arnaldo Antunes

RESUMO

Este trabalho realizou o porte de uma máquina virtual JAVA multi-thread para um sistema operacional orientado à aplicação. O principal foco foi a reutilização de componentes de um sistema operacional bem projetado em termos de engenharia de software e orientado à aplicação, assim evitando o código reescrito, compilado e carregado no sistema. Neste trabalho, as *threads* JAVA foram mapeadas diretamente para *threads* do sistema operacional, neste caso o EPOS, fazendo com que todo o controle de escalonamento continue sendo de responsabilidade do sistema operacional, que por sua vez segue a política de escalonamento de acordo com as configurações que lhe foram fornecidas. Portanto, a máquina virtual em questão manteve seu tamanho reduzido, pois tais funcionalidades não precisaram serem reimplementadas. Por fim, um estudo comparativo com outras máquinas virtuais *Java* para sistemas embarcados foi realizado.
Palavras-chave: JAVA, Maquina Virtual, *Thread*, Sistema Operacional, linguagens.

ABSTRACT

This work has made the port of a JAVA Virtual Machine to a Application-Driven operational system. The main focus was the reuse of components of a well designed and application-oriented operating system in terms of software engineering thus avoiding rewritten, compiled and loaded code in the system. In this work, the Java threads are mapped directly to operating system threads, in this case the EPOS, making the entire control scheduling remains the responsibility of the operating system, which in turn follows the scheduling policy according to the settings supplied to it. Therefore, the virtual machine in question has retained its small size, as these features did not have to be re-implemented. Finally, a comparative study was conducted with other virtual machines JAVA for embedded systems.
Keywords: JAVA, Virtual Machine, *Thread*, Operational System, languages.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

As aplicações para sistemas embarcados, em sua maioria, são escritas em C, C++, e assembly, no entanto estas linguagens não possuem gerenciamento automático de memória nativo, bem como, não abstraem os ponteiros do desenvolvedor. Dessa forma possibilitam que o programador acesse diretamente qualquer endereço de memória do sistema, podendo causar sérios danos ao sistema durante a execução de uma aplicação, dado que, qualquer segmento de memória (dependendo do sistema operacional somente endereços do processo podem ser acessados) pode ser acessado e alterado erroneamente. Além disso, a falta de um gerenciador de memória faz com que o desenvolvedor gaste um tempo significativo evitando que ocorra vazamento de memória na aplicação.

Considerando a complexidade e o aumento no tamanho dos códigos escritos para sistemas embarcados é altamente desejável um método garantindo portabilidade, produtividade e gerenciamento de memória (coletor de lixo). Bem como, é necessário abstrair do desenvolvedor a plataforma do sistema através da apresentação de uma interface de programação comum para uma grande variedade de plataformas. Para atingir tal objetivo, podem ser usadas máquinas virtuais como técnica de implementação de linguagens de alto nível (HLL, do inglês *High Level Language*). (GILOI, 1997).

Com a popularização do JAVA (LINDHOLM; YELLIN, 1999), que segundo Butter (2007), melhora a produtividade, facilita a manutenção dos códigos e possibilita um lançamento mais rápido do produto, faz com que a ideia de se ter uma máquina virtual (VM, do inglês *Virtual Machine*) rodando em sistemas embarcados seja atrativa. Além dos benefícios citados acima, soma-se a eles o surgimento da internet e da internet das coisas (IoT, do inglês *Internet of Things*) , que mudou a ideia de sistemas embarcados com funções fixas para sistemas mais abertos e com alguma forma de conexão em rede(ATZORI et al., 2010).

Uma técnica promissora para aumentar a produtividade e qualidade de aplicações para sistemas embarcados é, portanto, utilizar máquinas virtuais para implementação de linguagens de alto nível. Além disso, facilita a portabilidade do código executável, o que é de suma importância para a manutenção de micro-controladores usados em redes de sensores sem fio típicas(PAJIC; MANGHARAM, 2010).

1.1 MOTIVAÇÃO

Um problema de se ter máquinas virtuais em sistemas embarcados é o excesso de recursos necessários para a execução das mesmas. O principal motivo deste excesso ocorre do fato de que as maiorias das implementações destas máquinas reincidem muitas das funcionalidades já realizadas pelos sistemas operacionais. Como por exemplo, o escalonamento de *threads*, a troca de contexto e a sincronização destas.

Logo, com a utilização de um sistema operacional (SO) orientado à aplicação e projetado com base no paradigma orientado a objetos/componentes, a máquina virtual pode ser reduzida à apenas um interpretador de instruções da máquina virtual em questão, ou seja, a única função é interpretar e executar os códigos da máquina virtualizada. O sistema operacional deve fornecer todos os componentes que a máquina virtual necessita, ficando a cargo do sistema operacional o escalonamento das tarefas, gerenciamento de memória, e eventuais controles para sistemas de tempo real.

Contudo, a quantidade de memória necessária para o código executável do sistema operacional juntamente com a máquina virtual continuará reduzido, pois grande parte do código original é reutilizado, além do que, a máquina virtual, assim como o sistema operacional será orientada à aplicação.

1.2 OBJETIVOS

Considerando que os sistemas embarcados possuem restrições de memórias é necessário reduzir a dimensão do código gerado para realizar a implementação de uma máquina virtual nestes sistemas. Tendo em mãos este problema seguem os objetivos.

1.2.1 Objetivo Geral

Portar uma máquina virtual JAVA projetada para sistemas embarcados com severas restrições de memória, mais especificamente a NANOVM (HARBAUM, 2005), para o sistema operacional *Embedded Parallel Operational System* (EPOS). Reutilizar funcionalidades de um sistema operacional, como escalo-

namento de *threads* e troca de contexto.

1.2.2 Objetivos específicos

- Realizar o porte da NanoVM para o sistema operacional EPOS.
- Adaptar a NanoVM para o paradigma orientado a objeto.
- Fazer com que a NanoVM seja capaz de ter suporte à concorrência utilizando os mecanismos de escalonamento já implementados no sistema operacional EPOS.

1.3 METODOLOGIA

Por meio de técnicas de programação orientada a objetos, extendendo classes para criar especializações de componentes, é possível atingir os objetivos propostos.

O sistema operacional fornece os componentes que a máquina virtual necessita, mais especificamente as *threads*, que ao se tornarem *threads* JAVA são apenas objetos de uma classe que extende a *classe Active*, que representa as *threads* do EPOS.

Testes comparativos com outras máquinas virtuais projetadas para sistemas com restrições de memória, desempenho e energia são realizados para verificação de consumo de memória e desempenho.

1.4 ESTRUTURA DO TRABALHO

O restante do texto é organizado da seguinte forma, o capítulo 2 apresenta uma revisão teórica sobre máquinas virtuais e o estado da arte relacionado a este trabalho, assim como o sistema operacional utilizado neste trabalho. No capítulo 3, é apresentado o desenvolvimento do trabalho sendo seguido da apresentação dos resultados. Por fim, no capítulo 5 são feitas algumas conclusões sobre os resultados apresentados, assim como possíveis trabalhos futuros.

2 FUNDAMENTOS TEÓRICOS

Neste capítulo são descritos os conceitos fundamentais das máquinas virtuais como suporte para linguagens de programação, assim como uma visão geral de vários trabalhos realizados na área de máquinas virtuais para sistemas embarcados de significativa relevância. O sistema operacional EPOS e seus principais conceitos e filosofia de projeto também são abordados.

2.1 MÁQUINAS VIRTUAIS

Segundo (SMITH; NAIR, 2005), uma máquina virtual é implementada adicionando-se uma camada de software que tem como o objetivo fazer com que uma máquina real de suporte para a arquitetura da máquina virtual desejada. Formalmente, virtualização envolve a construção de um isomorfismo que realiza o mapeamento de um sistema virtual convidado (do inglês, *guest*) para um sistema real hospedeiro (do inglês, *host*) (POPEK; GOLDBERG, 1974). Este isomorfismo, ilustrado na figura 1, mapeia o estado do convidado para um estado equivalente do hospedeiro (função V na figura 1), e para uma sequência de operações *e*, que modifica o estado no convidado (a função modifica o estado *Si* para o estado *Sj*) existe uma sequência correspondente de operações *e'* no hospedeiro que desempenha uma modificação equivalente no estado do hospedeiro (muda de *Si'* para *Sj'*).

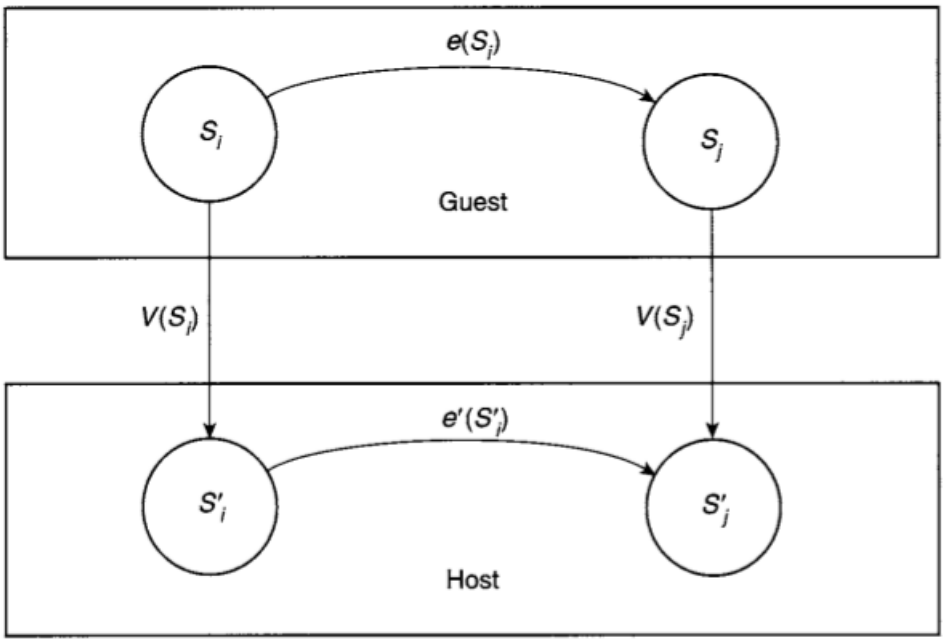

Figura 1 – Virtualização. Formalmente é a construção de um isomorfismo entre um sistema convidado e um hospedeiro; *e' o V(Si)=V o e(Si)*. Extraído de (SMITH; NAIR, 2005)

Devido aos benefícios oferecidos pela virtualização das máquinas, houve uma explosão no interesse relacionado às máquinas virtuais, principalmente por parte da indústria de software. Um dos principais motivos foi o surgimento da *World Wide Web*, e ainda, na primeira década do século XXI, houve um crescimento no interesse de se portar esta tecnologia para sistemas com restrições de memória, desempenho e energia. (SMITH; NAIR, 2005)

Dentro os benefícios oferecidos pela virtualização de máquinas podemos citar: isolamento, segurança, e balanceamento de carga. Por exemplo, um servidor de alto desempenho pode ser divido em múltiplos servidores menores para assim ser feito um controle de uso do hardware.

Já para sistemas embarcados, o principal benefício é que estas máquinas virtuais podem ser usadas como um método para assegurar portabilidade, assim como, para execução de linguagens independente da arquitetura real do sistema. Não obstante, lado elas oferecem oportunidades para enriquecer o ambiente de execução, e.g: gerenciamento de memória, proteção de memória, e também flexibilidade(CRAIG, 2006). Um exemplo de sucesso desta abordagem é a linguagem e máquina virtual JAVA.

As HLL VM (do inglês, *High-Level Language Virtual Machines*) (SMITH; NAIR, 2005) são geralmente definidas para programas com modo de operação usuário, e geralmente não desenvolvidas para um processador real, portanto só será executada num processador virtual. A interface com o sistema é feita através de um conjunto de bibliotecas (chamadas *"APIs"*) que podem acessar arquivos e realizar operações de rede por exemplo. Estas *"APIs"*, em geral, são implementadas com os chamados métodos nativos.

Os métodos nativos são mantidos com códigos binários nativos, como o nome sugere. Eles são úteis para executar funções do sistema hospedeiro, ou seja, chamadas do sistema operacional. No caso do JAVA, a interface nativa JAVA (*JNI*, do inglês Java/native interface) define uma convenção de como devem ser realizadas e implementadas estas funcionalidades nativas. Para a plataforma *Java Micro Edition* (JME) o nome dado é K *Native Interface* (KNI), projetada para a máquina virtual K - *K Virtual Machine* (KVM) (MICROSYSTEMS, 2012).

Os objetivos da JNI são: servir como uma interface comum para desenvolvedores de máquinas virtuais, para que as mesmas funções nativas irão funcionar sem modificações em diferentes máquinas virtuais e prover uma *API* no nível JAVA para que seja possível um programador JAVA carregar dinamicamente bibliotecas e acessar funções nativas contidas nestas bibliotecas. Infelizmente, por causa da natureza genérica, a JNI é bastante cara e introduz um *overhead* significante de memória e desempenho. Para a amenizar estes problemas a *KNI* defini um subconjunto da *JNI* que é apropriado para dispositivos com restrições de memória e energia. Sendo assim, os objetivos da *KNI* são: portabilidade de código nativo, desde que portados para o sistema operacional específico, isolamento entre funções nativas e detalhes de implementação da máquina virtual e ainda, eficiência de memória e *overhead* de desempenho mínimo.

Apesar da existência de máquinas virtuais desde os anos 60, foi com o advento do JAVA que elas se tornaram comuns como técnica de implementação de uma nova linguagem (CRAIG, 2006). Dentre algumas linguagens que já faziam uso desta técnica podemos citar: Prolog, Curry, Oz, Erlang.

No domínio de sistemas embarcados diversas são as abordagens utilizadas para tratar cada requisito de Sistema Embarcado. Para aumento de desempenho e redução do consumo de memória pode-se empregar o uso de máquinas virtuais dedicadas (BROUWERS et al., 2009; HARBAUM, 2005), técnicas de compilação e geração de sistema (THOMM et al., 2010; PIZLO et al., 2010) ou ainda o uso de hardware dedicado (PUFFITSCH; SCHOEBERL, 2007; SCHOEBERL, 2008). Utilizando-se de modelos de execuções previsíveis, análises de escalonabilidade podem ser executadas e requisitos de tempo real garantidos (BØGHOLM et al., 2009; ZERZELIDIS; WELLINGS, 2010; BØGHOLM et al., 2010). 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 et al., 2009).

2.1.1 Máquina Virtual Java - JVM

Com o slogan "Compile uma vez, e execute em qualquer lugar", a máquina virtual JAVA (JVM, do inglês, *Java Virtual Machine*) começou o interesse atual em máquinas virtuais. (CRAIG, 2006) A JVM é uma máquina baseada em pilha (*stack-based*) que possui 256 instruções. As principais características dela são:

- Tipos de dados primitivos
- Referências
- Objetos e *Arrays*
- Armazenamento de dados (global, local e operando)
- Pilha (*Stack*)
- Memória Global
- *Constant Pool*

Os dados primitivos (do ingês, *Primitive Data Types*) são os seguintes: *int* (inteiro), *char*, *byte*, *short*, *float*, *double* e *returnAddress*. O tipo *boolean* é implementado como um tipo primitivo *int* ou byte. Objetos podem ser compostos destes dados primitivos.

O tipo referência pode segurar valores de referência. Um valor de referência aponta para um objeto salvo em memória. Uma referência pode ter o valor especial *null* (indefinido) caso não tenha sido atribuído um valor para a mesma.

Objetos carregam dados guardados numa estrutura lógica declarada pelo programador. Estes são compostos por dados primitivos e referências que podem apontar para outros objetos. Um *Array* é um tipo de objeto especial com um conjunto de instruções explícitas. Para cada *array* é definido, no momento em que é instanciado, um número fixo de elementos que não muda durante a execução do programa. Os elementos de um *array* devem ser todos do mesmo tipo primitivo ou devem ser todos referências. No caso de serem referências, então, devem apontar para objetos do mesmo tipo.

Na máquina virtual JAVA existem três tipos de armazenamento de dados, são eles: global, local e operando. Armazenamento global é a memória principal, onde as variáveis globais residem. Armazenamento local é um armazenamento temporário para variáveis que são locais para um método. Armazenamento de operando guarda variáveis enquanto elas estão sendo operadas por instruções funcionais (aritméticas, lógicas e *shifts*).

A arquitetura da memória principal na JVM contém uma área de método (*method area*) que guarda o código e um armazenamento global para salvar *arrays* e objetos. O espaço da memória global é gerenciado como uma *heap* de tamanho não especificado, ou seja, o tamanho depende da implementação e não da especificação. A *heap* pode guardar tanto objetos dinâmicos quanto estáticos, incluindo *arrays*; estes objetos são criados na *heap* em tempo de execução. Quando um objeto é dinamicamente criado na *heap*, uma referência é gerada para apontar para este objeto. Objetos na *heap* só podem ser acessados através de referências que possuem um tipo que corresponde ao tipo do objeto.

Armazenamento local e de operando são alocados na pilha, assim como os argumentos de métodos. As instruções da máquina nunca podem colocar *arrays* ou objetos na pilha; somente referências e elementos individuais podem ser colocados nestas. Para cada método chamado, um *stack frame* é alocado (figura **??**) com argumentos, armazenamento local (variáveis locais) e armazenamento de operando, nesta ordem. O armazenamento local de um determinado método é de tamanho fixo; o tamanho propício de espaço da pilha necessário para armazenamento local pode ser determinado durante a compilação.

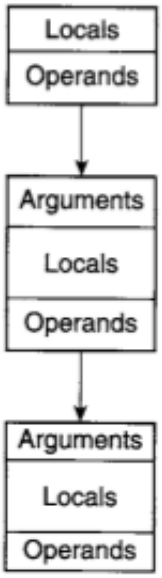

Figura 2 – Estrutura da pilha da máquina virtual JAVA

A arquitetura da memória principal na JVM contém uma área de método (*area method*) que guarda o código e um armazenamento global para salvar *arrays* e objetos. O espaço da memória global é gerenciado como uma *heap* de tamanho não especificado, ou seja, o tamanho depende da implementação e não da especificação. A *heap* pode guardar tanto objetos dinâmicos quanto estáticos, incluindo *arrays*; estes objetos são criados na *heap* em tempo de execução. Quando um objeto é dinamicamente criado na *heap*, uma referência é gerada para apontar para este objeto. Objetos na *heap* só podem ser acessados através de referências que possuem um tipo que corresponde ao tipo do objeto.

Dados constantes associados a um programa são colocados num bloco conhecido como *constant pool*. Qualquer instrução, que necessite de valores constantes, pode obter os mesmos através de busca por índice na *constant pool*. Isto faz com que as instruções da JVM fiquem mais compactas e uniformes.

A figura 3 ilustra a hierarquia de memória na JVM. O gráfico mostra que um *array* foi alocado na *heap*, e a referência para o mesmo faz parte de outro objeto. Note que no canto direito inferior da *heap*

existe um objeto que não possui nenhuma referência apontando para ele. Logo, este é um objeto pronto para ser coletado pelo coletor de lixo.

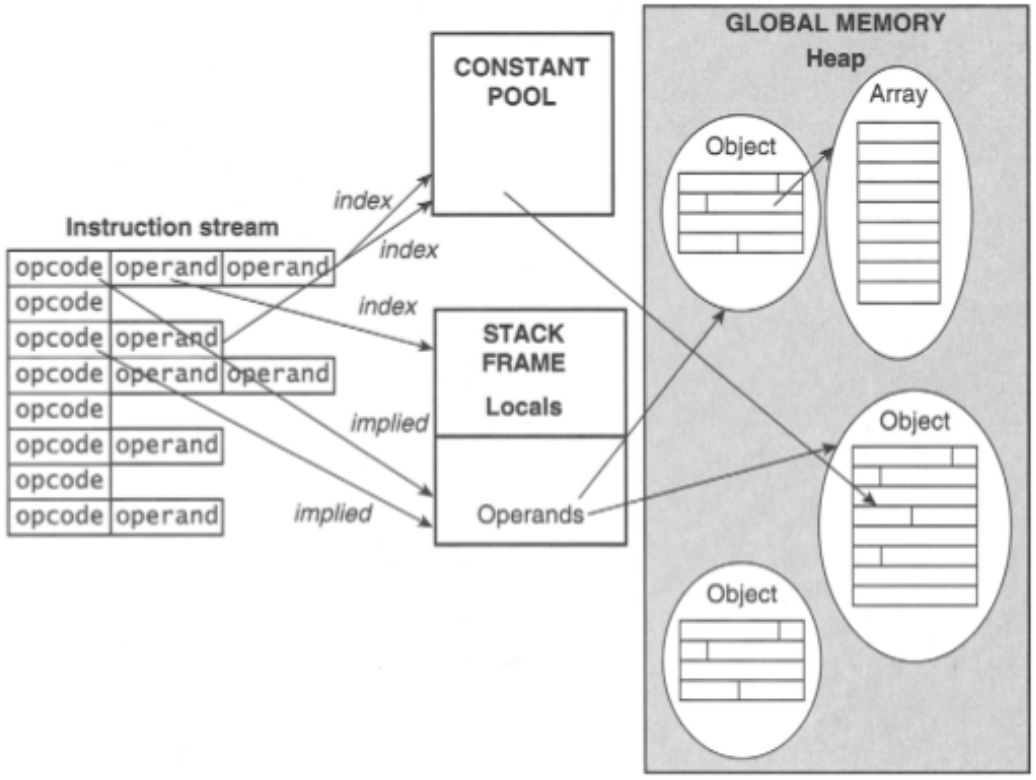

Figura 3 – Hierarquia de memória usada por um programa Java. Extraído de (SMITH; NAIR, 2005)

Todas as instruções da JVM contem um byte *opcode* e zero ou mais bytes subsequentes, dependendo do *opcode*. *Opcode* é o código que representa uma instrução da JVM. A figura 4 mostra os formatos das instruções da JVM. Cada campo da instrução contém exatamente um byte, no entanto um ou mais campos podem ser concatenados para formar um único operando. O campo *index* é usado como índice na *constant pool* ou áreas de armazenamento local. O byte *data* pode ser um dado imediato ou um *offset* para um salto relativo do *PC (Programa Counter)*.

Os *bytecodes* ficam armazenados no *method code vectors* localizados nos arquivos que representam as classes (*class files*). Estas instruções podem ser divididas em duas classes:

- Instruções simples. *Jumps* e instruções aritméticas.
- Instruções complexas. Realiza alocação de instâncias de classe e *arrays*, acesso aos *arrays*, acesso e atualização de instâncias de objetos, controle de exceções, e instruções para invocar métodos de todos os tipos.

Na figura 5 podemos ver a disposição de uma classe binária, ou seja, a estrutura de uma classe JAVA. O formato da classe binária é, de fato, a interface suportada pela máquina virtual. Alguns campos apresentados na figura que merecem destaque são:

- *Magic Number* é uma sequência de caracteres que é a mesma para todas as classes binárias JAVA.
- *Constant Pool* guarda todos os valores e referências constantes usados pelos métodos.
- *Access Flags*, como o nome sugere, provem informações sobre o acesso, ou seja, se uma classe em particular é pública ou privada, ou se é uma interface e não uma classe.
- *This_Class e Super_Class* contém os nomes desta classe e da superclasse desta classe. Ambos são fornecidos como índices para a *constant pool*. Os nomes estão na *constant pool*.
- *Interface* contém um número de referências para as *superinterfaces* desta classe, isto é, a interface pela qual esta classe pode ser diretamente acessada. Estes também são índices na *constant pool*. As entradas na *constant pool* são referências para as interfaces.

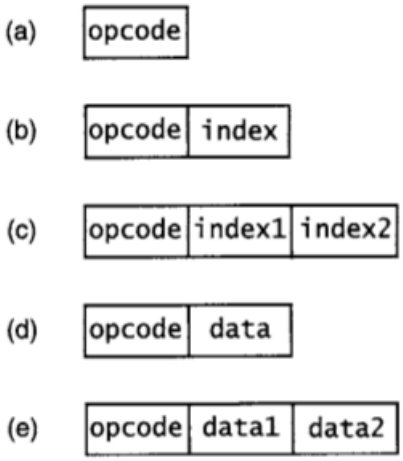

Figura 4 – Formato típico de instruções *bytecode*. Extraído de (LINDHOLM; YELLIN, 1999)

- *Fields* contém a especificação dos campos declarados para esta classe. Esta informação é incluída numa pequena tabela para cada campo; a tabela inclui informação de acesso (público, privado, protegido), um índice de nome (*offset*) para a entrada da *constant pool* que contém o nome deste campo, um índice descritor (o qual contém um índice para a *constant pool* onde o descritor para este campo pode ser achado), e informações de atributos.
- *Methods* contém a informação pertencente a cada método, ou seja, o nome e descritor, assim como o método em si, codificado como um fluxo de *bytecodes*. Cada método também pode ter tabelas de atributos, por exemplo, dando a máxima profundidade da pilha de operando para o método e o número de variáveis locais.
- *Attributes* contém informações detalhadas dos componentes listados anteriormente.

Uma visão geral de como funcionam as chamadas nativas na JVM (JNI) pode ser vista na figura 6. No lado esquerdo temos o lado JAVA, do lado direito temos o código compilado para a plataforma nativa. Cada lado da figura compila para o seu próprio modo binário. No lado do JAVA, estes são classes binárias padrões; no lado nativo eles são programas binários na forma nativa da plataforma. Os dados do lado JAVA existem como objetos e *arrays* na *heap* e variáveis na pilha. No lado nativo, os dados são organizados pelo compilador nativo. Como mostrado na figura, a JNI provê uma maneira de um método JAVA invocar um método nativo. Para isto basta que o método seja declarado com a palavra reservada *native*, assim este vira um método que será executado através de código binário da plataforma específica.

As chamadas nativas, em sua maioria, são utilizadas para chamar funcionalidades do sistema operacional e utilizam uma pilha própria. Esta pilha não faz parte da JVM. Em geral, os objetivos destes métodos é prover à JVM acesso à sistemas gráficos do *host*, e.g: X Windows, conexão, e.g: *sockets*, etc.

2.1.2 Java para sistemas embarcados

A linguagem de programação JAVA foi originalmente desenvolvida para facilitar o desenvolvimento de dispositivos eletrônicos de consumo (ORTIZ, 2007). No entanto com a explosão da WEB a linguagem JAVA acabou por ser adaptada para resolver os problemas gerados pela WEB. Mas na última década, de diversas maneiras, a plataforma JAVA está retornando às origens da tecnologia Java.

A máquina virtual edição micro, doravante chamada de JME, define um conjunto de ambientes de execução e *APIs* para sistemas embarcados, como telefones móveis, *TV set-top boxes*, e outros dispositivos que têm restrições para suportar uma JVM SE completa. .

A JME não define uma nova linguagem. Em vez disso, adapta a tecnologia JAVA já existente para sistemas embarcados. Esta é conceptualmente organizada como uma pilha consistindo de quatro camadas de softwares como mostra a figura 7. A camada de configuração consiste da máquina virtual e *core Java APIs*. Acima da configuração estão os *profiles* e o ambiente da aplicação, o qual consiste de APIs e ambientes de execução como o *Mobile Information Device Profile*, *Java Technologies for Wireless Industry* e *Mobile Service Architecture*.

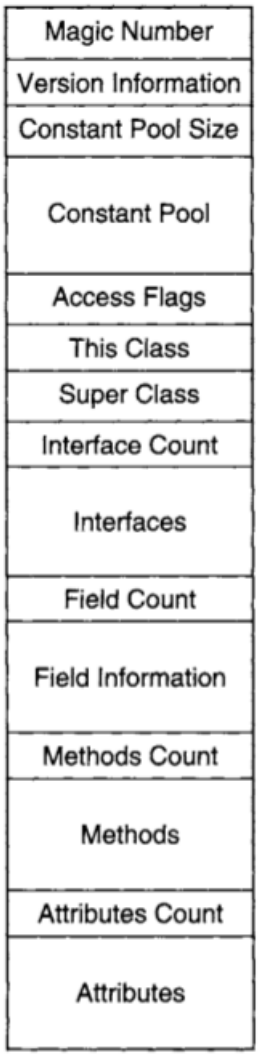

Figura 5 – Formato de uma classe binária Java

Saindo da especificação JME e olhando para as pesquisas atuais na área acadêmica, segundo (VITEK, 2011), as implementações da JVM para sistemas embarcados podem ser separadas em três classes:

- **Interpretadas**: Tipicamente lidam com *bytecode* diretamente, interpretadores são lentos, mas com razoável eficiência de memória, além de suportar carregamento dinâmico de classes.
- **Compiladas no momento**: Geram o código executável no momento da execução. Variam no desempenho, dependendo do nível de otimização aplicado ao *bytecode*. Tipicamente utilizam mais memória e podem introduzir pausas durante a execução do sistema.
- **Compiladas à frente do tempo**: Podem gerar código altamente otimizado, compilando para C ou diretamente para código nativo. Geralmente não suporta carregamento de classe dinâmico.

2.2 EPOS

O EPOS (FRÖHLICH, 2001) (*Embedded Parallel Operating System* é um sistema operacional que segue os princípios da metodologia ADESD *Application-driven Embedded System Design* (FRÖHLICH, 2001) , ou seja, gera-se um sistema individualizado para a aplicação alvo.

A ADESD guia a concepção e o desenvolvimento de sistema em embarcados dedicados, os quais executarão uma única aplicação. No entanto, 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

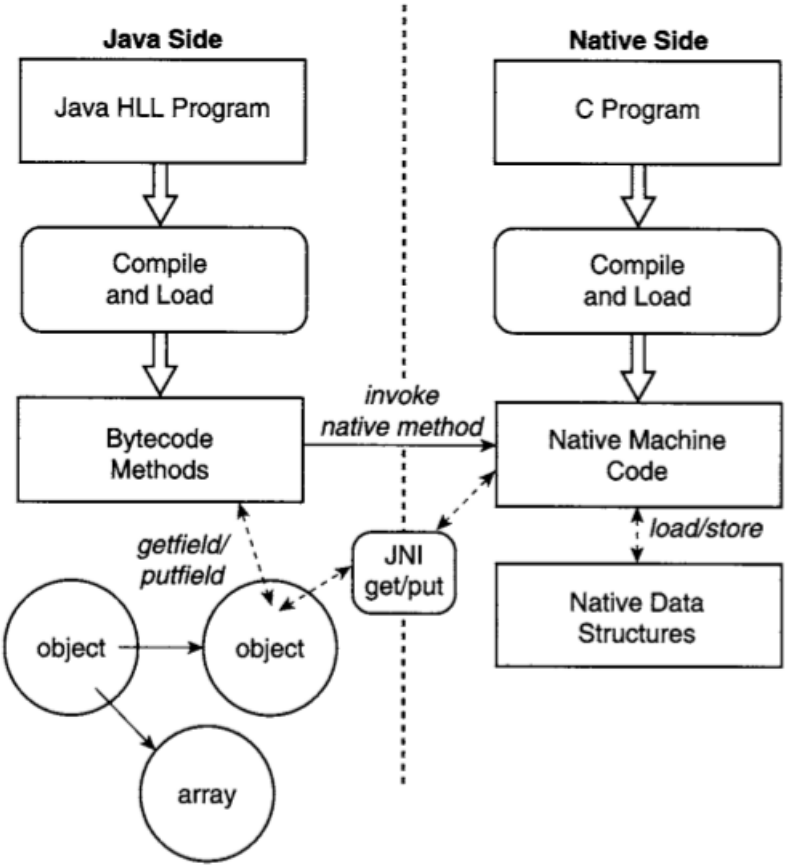

Figura 6 – *Java Native Interface*. Permite a chamada de funcionalidades do sistema operacional, e inter-operação entre código JAVA e código nativo. Extraído de (SMITH; NAIR, 2005)

pertencentes a um mesmo domínio(FRÖHLICH, 2001). Na figura 8 é possível ter uma visão geral de um sistema orientado à aplicação. Esta figura mostra como o domínio é decomposto em abstrações e estas abstrações agrupadas em famílias de acordo com as similaridades entre elas.

A configuração do sistema, característica marcante de um sistema operacional orientado à aplicação, neste caso o EPOS, é atingido através da utilização de várias técnicas, entre elas podemos citar famílias de abstrações do sistema, *frameworks*, aspectos, etc. As abstrações são implementações de conceitos clássicos e pertinentes ao domínio de sistemas operacionais, como por exemplo, *Threads*, Semáforos, Escalonadores, entre outros. As abstrações são independentes de aplicação, permitindo uma grande variedade de sistemas gerados, visto que são reusáveis.

No contexto deste trabalho, a abstração correspondente à parte ativa de um processo (*Thread*) e a sincronização destas partes foram usadas integralmente. Descrições detalhadas de como são organizadas e implementadas estas abstrações são encontradas em (FRÖHLICH, 2001).

Em se tratando de portabilidade, a solução adotada no projeto do EPOS foi o uso de mediadores de hardware (POLPETA; FRÖHLICH, 2004). Estes são abstrações de componentes de hardware (e por isso, sua implementação é específica para cada máquina), que fazem o interfaceamento entre o sistema operacional e estes componentes. Através deste mecanismo, se permite que a comunicação entre o sistema operacional e o hardware seja centralizada nestes mediadores, e que dependências arquiteturais não se espalhem pelo sistema. Atualmente o EPOS suporta as seguintes arquiteturas:

- AVR8
- ARM7
- IA32(Intel x86)
- PowerPC
- MIPS

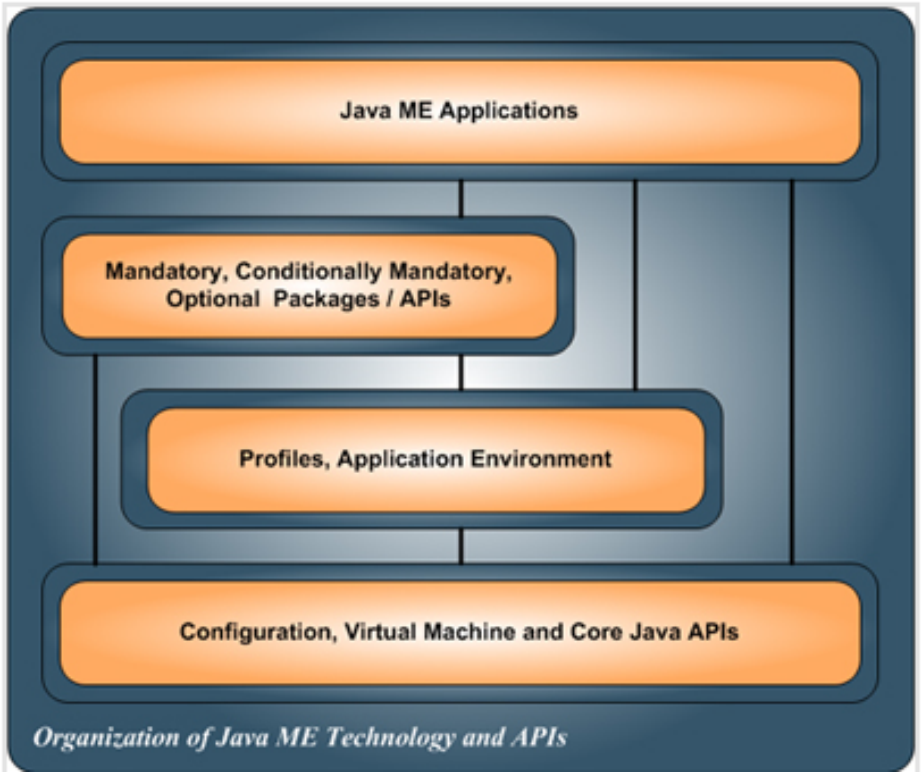

Figura 7 – Organização da JME e APIs. Extraído de (ORTIZ, 2007)

Além disso, o sistema operacional foi projetado para fazer uso eficiente de recursos, permitir a execução de aplicações dedicadas de alto desempenho, ser configurável de acordo com as necessidades da aplicação e facilmente portável.

Características como uso eficiente de memória e energia, portabilidade (tanto em arquiteturas portadas quanto em facilidade de porte), extensão e criação de componentes, suporte a aplicações de alto desempenho, entre outras, são fatores que foram levados em consideração para a solução proposta. Neste contexto o EPOS se apresentou como uma boa solução, pois possui essas características.

2.3 TRABALHOS RELACIONADOS

A partir do ano 2000 o interesse em portar máquinas virtuais para sistemas embarcados tem aumentado consideravelmente. Os principais motivos são produtividade, reusabilidade. No caso do JAVA, a disponibilidade de pessoal qualificado (VITEK, 2011). As seguintes máquinas tiveram uma importância considerável.

2.3.1 Darjeeling

A máquina virtual DARJEELING (BROUWERS et al., 2009), modelada de acordo com a especificação da JVM, é capaz de executar um subconjunto considerável da linguagem Java, no entanto, foi desenvolvida para rodar especificamente em micro-controladores de 8 e 16 bits e com 2-10 KB de RAM.

Segundo os autores, a DARJEELING foi a primeira máquina virtual Java para redes de sensores sem fio de código aberto (*open source*). A mesma além de possuir uma rica quantidade de características da JVM , como *light-weight threads*, controle de memória dinâmico (coletor de lixo), e controle de exceção, é otimizada para micro-controladores com restrição de memória. O objetivo é alcançado combinando o compilador Java tradicional, um analisador e transformador de *bytecodes off-line*, uma ambiente de execução com instruções sobre medida para uma arquitetura de 16 bits, e um novo modo de organização de memória que separa referências dos tipos primitivos, assim eliminando o tempo de execução necessário durante a verificação de tipo realizada pelo coletor de lixo. Na figura 9, retirada de (BROUWERS et al., 2009) podemos notar a diferença entre as pilhas da JVM e da Darjeeling.

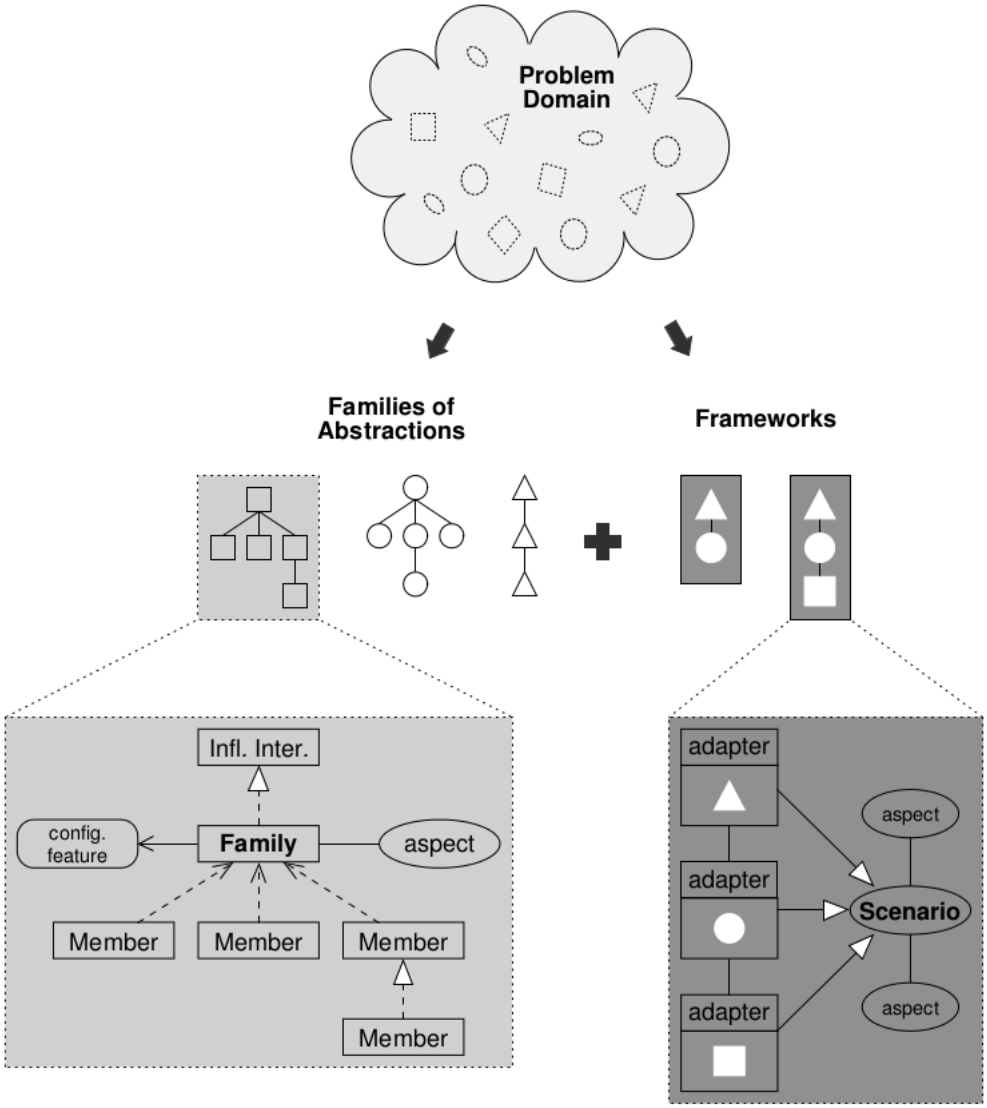

Figura 8 – Visão geral de um sistema orientado à aplicação. Extraído de (FRÖHLICH, 2001)

Além disso, a DARJEELING se situa na classe de máquinas virtuais genéricas, diferente de outras máquinas que são classificadas como máquinas virtuais para aplicações específicas (ASVM). Importante notar que a DARJEELING foi desenvolvida para interpretar *bytecodes*, assim como a máquina proposta neste trabalho.

2.3.2 Maté

O principal foco da máquina virtual MATÉ (LEVIS; CULLER, 2002) é possibilitar a transmissão de aplicativos para os motes de uma rede de sensores sem fio, ou seja, instruções para serem decodificadas e executadas pelo interpretador de *bytecodes* MATÉ instalado no sensor. Para isto se tornar viável é necessário lidar com a energia gasta durante o envio destes pacotes que contem os *bytecodes*.

Esta máquina roda sobre o sistema operacional TINYOS (TINYOS, 2012). Um aspecto interessante é que a MATÉ esconde o assincronismo do TINYOS. Os autores desta máquina alegam que o modelo síncrono de programação faz com que o desenvolvimento de aplicações se torne mais simples e também que existe uma menor possibilidade de ocorrerem erros (*bugs*) do que lidar com notificações de eventos assíncronos.

A contribuição mais importante desta máquina é a nova formatação dos códigos, ou seja, a simbologia das instruções que a máquina deve decodificar e executar. Na figura 10 podemos ver como estas instruções estão codificadas e, fácil notar, a economia de espaço de memória necessário para transmitir os

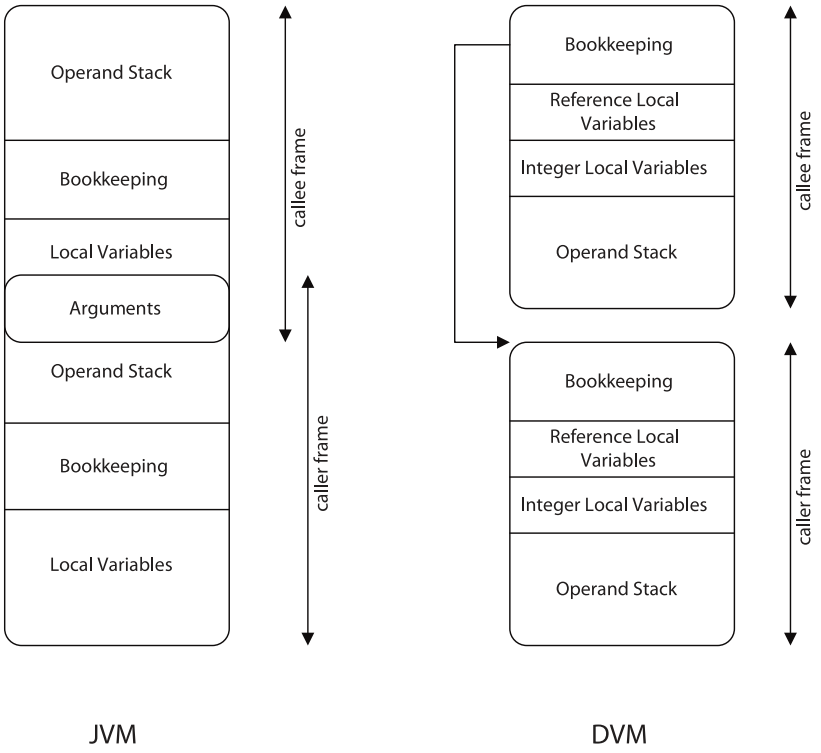

Figura 9 – Uma comparação do layout da pilha na Darjeeling, extraído de (BROUWERS et al., 2009)

códigos pela rede de sensores sem fio. Cada pacote de mensagem do TINYOS suporta até 24 instruções MATÉ.

basic	00iiiiii	i = instruction
s-class	01iiixxx	i = instruction, x = argument
x-class	1ixxxxxx	i = instruction, x = argument

Figura 10 – Formato de instruções da MATÉ

2.3.3 KESO

A KESO(THOMM et al., 2010) é outra máquina virtual baseada na JVM e sobre a licença LGPL. Diferentemente da proposta da MATÉ que tem como principal foco aplicações para redes de sensores sem fio e que podem mudar de objetivo a qualquer momento, a KESO foca no domínio de sistemas embarcados com aplicações estáticas. Além disso, o desenvolvimento da mesma teve como um dos principais focos o domínio de sistemas embarcados com restrições de tempo real, se somando a este a preocupação com proteção de memória e isolamento espacial de componentes de software ou diferentes aplicações que co-habitam um micro-controlador.

Para o coletor de lixo funcionar como assumido pela KESO, ou seja, com garantias de tempo real, requer que o escalonamento de tarefas seja realizado baseado em prioridades, mecanismo que qualquer sistema operacional de tempo real deve ter. Para acessar funcionalidades do sistema operacional a *KESO* utiliza de uma interface de chamadas nativas (*KESO Native Interface* (KNI). .

Diferentemente das máquinas já citadas, a KESO não interpreta *bytecodes* JAVA. Ela realiza o que é chamado de compilação a frente do tempo (*Ahead-of-Time Compilation*), ou seja, toda a aplicação escrita em JAVA passa por um processo de transformação e torna-se código nativo. No caso da KESO, antes de ser realizada esta transformação para o código tornar-se nativo, o compilador *jino* gera um código ISO-C90, e então é usado um compilador da linguagem C para gerar o código nativo. A máquina virtual FijiVM (**??**) também pode ser citada como uma máquina com compilação a frente do tempo.

Esta máquina conseguiu manter o *overhead* do tamanho do código de uma aplicação escrita em JAVA em relação a uma escrita em C em menos de 10%. Utilizando de chamadas nativas para utilizar funções da biblioteca o *overhead* de execução também não passou de 10%.

2.3.4 NanoVM

A NanoVM(HARBAUM, 2005) é uma JVM, escrita em C, para o *Atmel AVR Atmega8 CPU*, um membro da família *AVR*. Esta máquina surgiu com o objetivo de facilitar a programação do *DLR Asuro Robot*. O principal foco da implementação desta máquina é a pouca utilização de memória, e como consequência não implementa muitas das funcionalidades descritas na especificação da JVM. A mesma necessita apenas de 8 *Kbytes* de memória *flash* para ser inicializada, e ainda contém algumas classes nativas, que realizam as chamadas de baixo nível, como por exemplo, configurações de GPIO . Algumas das características da NanoVM:

- Suporte para *bytecode*
- Aritmética de inteiros de 15/21 bit (Configurável)
- Opção para operações matemáticas utilizando ponto flutuante
- Coletor de lixo
- Suporte à herança
- Arquitetura unificada de *heap* e pilha

Com a abstração do hardware fornecida pela NanoVM, o desenvolvedor não precisa se preocupar para qual o tipo de micro-controlador o código está sendo desenvolvido. O mesmo compilador JAVA (*javac*) e a ferramenta *NanoVMTool* podem ser usados para quaisquer sistemas rodando em qualquer tipo de micro-controladores suportados por esta máquina.

As chamadas nativas da NanoVM são relativamente simples de serem adicionadas e implementadas. Para isto basta criar os arquivos de configurações para a NanoVMTool. O mesmo deve possuir as assinaturas dos métodos também. Na NanoVM não é possível misturar numa classe métodos nativos com métodos JAVA, se uma classe é declarada nativa, todos seus métodos são nativos. Sendo podemos dizer que a NanoVM possui classes nativas e não métodos nativos.

Uma ferramenta importante que faz parte da NanoVM é o *NanoVMTool*. Tal ferramenta, escrita em JAVA, realiza a otimização e adaptação do fluxo de *bytecodes* gerados pelo compilador JAVA padrão (*javac*).

2.4 SUMÁRIO

Analisando as propostas de máquinas virtuais para sistemas embarcados existentes, ou pelo menos de conhecimento deste grupo, podemos notar que existe espaço para uma nova proposta. Apesar de existirem máquinas virtuais orientadas a aplicação, nenhuma delas tem como base a especificação da JVM. E as máquinas que tem como fundamento a especificação da JVM além de não serem orientadas a aplicação ocupam memória demasiadamente.

Outro ponto interessante da proposta apresentada neste trabalho, é o fato de aproveitar ao máximo os componentes de uma sistema operacional orientado a aplicação. Isto faz com que a máquina virtual seja, quase que naturalmente, uma máquina virtual orientada a aplicação.

Por fim, a máquina virtual NanoVM foi escolhida como um dos alicerces do trabalho proposto por ser uma máquina virtual capaz de interpretar *bytecodes* e enxuta. Portanto, facilitando a decomposição da NanoVM para que somente o decodificador de *bytecodes* da NanoVM seja utilizado. Assim o interpretador de *bycodes* será um componente do sistema operacional. Logo, a máquina virtual terá portabilidade para todas as arquiteturas de processadores que o sistema operacional suporta, sem ser necessário escrever uma linha de código.

3 DESENVOLVIMENTO

A integração de uma máquina virtual com um sistema operacional, proposta neste trabalho, objetiva tornar a máquina virtual e o sistema operacional um só. Integrando a máquina virtual ao sistema operacional são eliminadas camadas intermediárias e a máquina virtual, antes vista como um *middleware* completo, passa a ser apenas um interpretador de *bytecodes*

Com um sistema operacional orientado à aplicação a máquina virtual também o será. E.g.: Se a aplicação não precisar de suporte a comunicação, as classes que implementam abstrações de comunicação (e.g. Network) não estarão presentes no sistema operacional e, consequentemente, também não estarão na máquina virtual.

Como pode ser visto na figura 11 a máquina virtual é apenas um componente pequeno do sistema operacional, como dito anteriormente, apenas um interpretador de *bytecodes*.

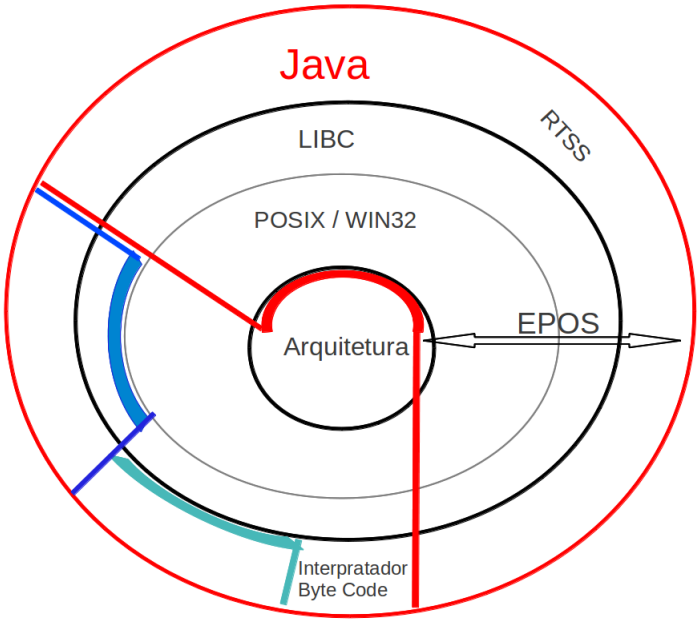

Figura 11 – JVM é apenas um interpretador do SO EPOS

3.1 IMPLEMENTAÇÃO

Para uma integração da NanoVM com o sistema operacional EPOS, conforme a proposta descrita acima, foi necessário não apenas adicionar componentes à mesma, como por exemplo as *threads*, mas também modificar o núcleo da máquina virtual em questão, sendo mais específico, a pilha da máquina virtual. Além disso, a NanoVM original foi implementada na linguagem de programação C, sendo necessária uma adaptação da mesma para o paradigma orientado a objetos e consequentemente para a linguagem de programação C++.

O principal motivo para se realizar a adaptação da NanoVM para o paradigma orientado a objeto foi facilitar a integração da máquina virtual com o sistema operacional EPOS (que é implementado utilizando o paradigma orientado a objeto e é escrito em C++). Desta forma permite-se assim uma integração mais efetiva entre componentes da NanoVM e componentes do EPOS, através da utilização dos mecanismos de heranças. Apesar do compilador GCC (GOUGH; STALLMAN, 2004) compilar tanto programas escritos em C quanto C++, existe uma grande diferença em termos conceituais quando se deseja desenvolver uma aplicação utilizando o paradigma orientado a objeto. Além disso, utilizando corretamente este paradigma é possível usufruir ganhos gerados por uma linguagem orientada a objetos(LEWIS et al., 1991) em relação a uma linguagem estruturada.

A NanoVMTool, responsável pela compactação dos *bytecodes* gerados pelo compilador da linguagem JAVA, não sofreu nenhuma modificação. Assim como as chamadas de métodos nativos, que permanecem com o mesmo mecanismo utilizado pelo NanoVM original.

3.2 CHAMADAS NATIVAS

As funcionalidades do sistema operacional podem ser utilizadas por meio da implementação de classes JAVA nativas. Estas, por sua vez, invocam métodos nativos, ou seja, métodos que realizam chamadas do sistema. Utilizando-se desta técnica foram criados *wrappers* na implementação da máquina virtual que adaptam funções do sistema operacional para a máquina virtual. Estes *wrappers*, em sua essência, são classes que herdam classes do EPOS.

Através do conceito de herança, fornecida por linguagens de programação que adotam o paradigma orientado a objeto, as funcionalidades do sistema operacional não precisam ser reimplementadas. Basta que sejam criados *wrappers* que adicionam membros e métodos conforme a necessidade da máquina virtual. Portanto, a máquina virtual apenas interpretará os *bytecodes* e quando for necessário utilizar alguma das funcionalidades do sistema operacional, o respectivo *wrapper* criado será o responsável. Este deve fazer o mínimo necessário para adaptar a funcionalidade do sistema operacional para a máquina virtual. Neste contexto o mínimo necessário significa não reescrever nenhuma funcionalidade que tenha sido implementada no sistema operacional. Estas funcionalidades devem ser utilizadas através da herança da classe pai.

Os passos necessários para uma chamada nativa são demonstrados na figura 12.

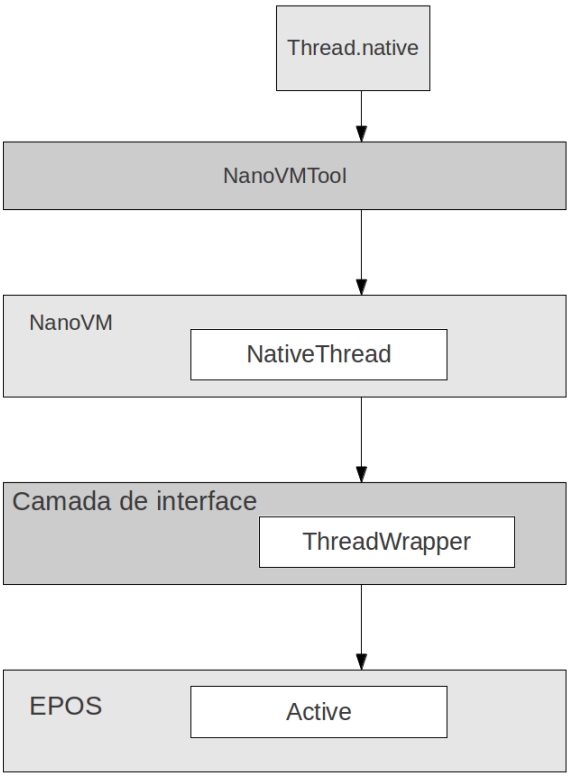

Figura 12 – Passos necessários para a comunicação de um código JAVA com a plataforma nativa

3.3 SUPORTE A *MULTI-THREAD*

O suporte a *multi-thread* foi realizado com chamadas nativa do sistema e utilizando o conceito de *wrapper* descrito acima. Assim, cada instância de um objeto JAVA, pertencente à classe *Thread*, é

mapeada diretamente para uma *thread* do sistema operacional. Tal mapeamento foi realizado utilizando o identificador do objeto JAVA.

À classe adicionada à máquina virtual foi dado o nome de *ThreadWrapper*. Esta extende a classe *Active* do EPOS, a qual representa as *threads* do sistema operacional. A estrutura desta classe e sua relação com outros componentes é mostrada no diagrama de classes UML mostrado na figura 13.

A classe *ThreadWrapper* além de possuir o atributo *program counter* (PC) , possui o método privado *vm_run* responsável pela interpretação e execução das instruções JAVA, os *bytecodes*. Todavia, todos os outros métodos são realizados através do mecanismo de herança. Ou seja, a troca de contexto que deve ser realizada quando existe uma alternância de thread a ser executada no processador fica a cargo do sistema operacional. Além disso, todo o processo de escalonamento de tarefas também é realizado diretamente pelo sistema operacional.

Apesar de ser necessário total confiança no mecanismo fornecido pelo sistema operacional, a máquina virtual não precisa reimplementar estas funcionalidades, a mesma foi herdada da classe pai, neste caso a classe *Active*. Sendo assim, com uma pequena quantidade de código adicionado foi possível incluir o mecanismo de múltiplas tarefas numa máquina virtual JAVA para sistemas embarcados que, originalmente, não possua tal mecanismo.

É importante notar que todas as threads da máquina virtual são iguais, ou seja, a thread principal, também chamada de (*main*), não possui nada de diferente em relação às outras. Portanto o método de entrada de um programa JAVA é executado exatamente como se fosse um método *run* de um objeto pertencente à classe Thread.

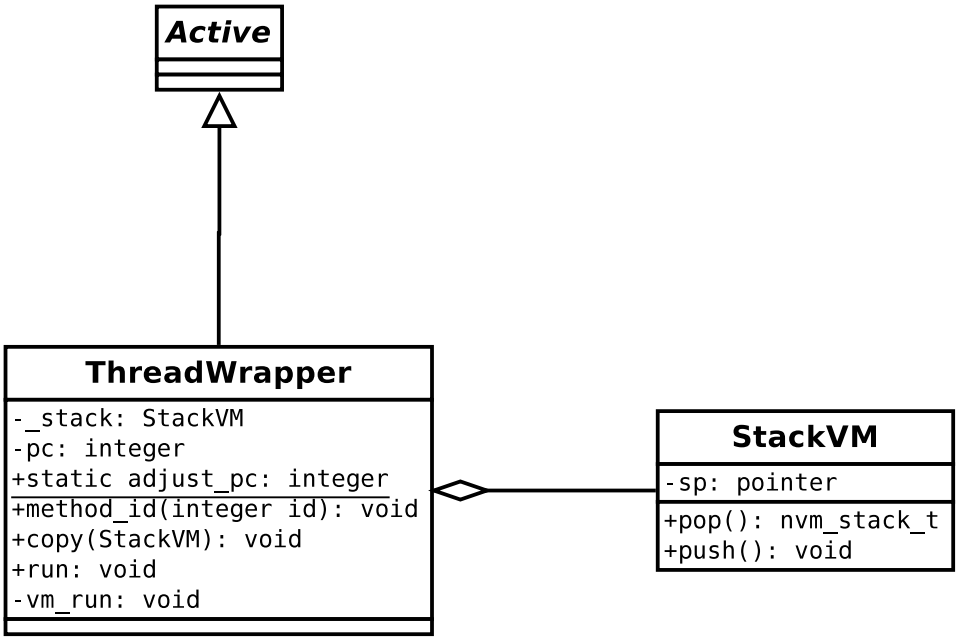

Figura 13 – Diagrama UML da classe ThreadWrapper

3.3.1 Criação de uma nova *thread*

Quando uma nova *thread* JAVA é instanciada, a máquina virtual NanoVM aloca este novo objeto na *heap*, juntamente como seu identificador, exatamente como se fosse qualquer outro objeto *Java*. Isto é realizado pela chamada do método *vm_new*.

Como é mostrado no diagrama de sequência da figura 14 após as funções tradicionais da máquina virtual, é executado o método *thread_init*. A invocação deste método é realizada pela função da máquina virtual responsável por chamar o respectivo construtor nativo de um objeto pertencente à uma classe nativa. Em outras palavras, o construtor pai deste objeto pertencente à classe *Thread* é nativo. Este, por sua vez, é responsável por criar uma *thread* do sistema operacional e, com o identificador do objeto JAVA, fazer o devido mapeamento.

Com o objetivo de manter a semântica das *threads* JAVA tradicionais, a mesma é construída no estado suspenso. Sendo necessário o objeto invocar o método *start* para que assim o estado da mesma seja

modificado para *ready*. Em suma, a *thread* estará pronta para ser executada, de acordo com a política de escalonamento adotada pelo escalonador do sistema operacional após a execução do método *start*.

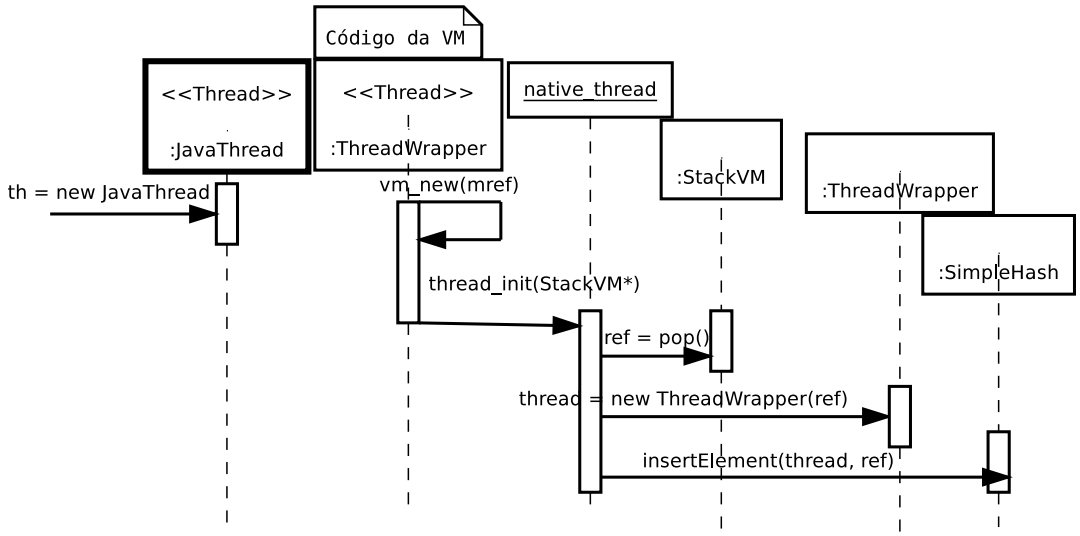

Figura 14 – Construção de um objeto *Java* pertencente à classe Thread

3.3.2 A inicialização da *thread*

A figura 15 ilustra a sequência dos métodos necessários para a inicialização de uma *thread* JAVA. Com a invocação do método *start*, de um objeto que pertence à classe *Thread*, a máquina virtual realiza uma chamada de sistema. Neste instante, o sistema operacional realiza as operações necessárias para deixar em modo *ready* a respectiva *thread* do sistema operacional.

Utilizando o identificador deste objeto é possível recuperar a referência da *thread* do sistema operacional EPOS, devido ao mapeamento feito durante a construção do objeto em questão. Na sequência é invocado o método privado responsável por ajustar o atributo *PC* das *threads* sendo aqui discutidas. Explicação mais detalhada sobre este método é encontrada na seção seguinte.

Através da invocação do método *method_address (id)*, a *thread* que está prestes a ser colocada em modo de execução (estado *ready*), recebe o identificador do método que deve ser executado por ela. Este identificador serve para que a *thread* execute a sequência de *bytecodes* gerados pelo método *run* da classe a qual o objeto pertence, ou seja, o segmento de código que representa o método *run*. E utilizando-se deste identificador, é possível posicionar o PC exatamente no *bytecode* que representa o início deste método na sequência de *bytecodes* que é a aplicação JAVA compilada.

Por fim, uma cópia das referências dos objetos que estão na *heap* da máquina virtual é passada para a pilha da *thread* em questão. Na sequência o método *start* da *thread* do sistema operacional EPOS é invocado. Deste momento em diante o controle desta *thread* JAVA é feito pelo sistema operacional. A máquina virtual não tem nenhuma responsabilidade sobre as trocas de contextos realizadas com a utilização dos registrados da máquina real. Assim como todo o mecanismo de escalonamento, que é gerenciado pelo escalonador sistema operacional.

3.3.3 Ajustar o *Program Counter*

Durante a execução dos métodos que precisam ser invocados para passar a *thread* do estado *suspended* para o estado *ready*, o método *adjustPC* é chamado. A figura 16 ilustra a sequência de métodos executados para a que esta função seja realizada.

A função deste método é fazer com que a *thread* que está sendo executada neste instante tenha seu o *Program Counter* (PC) ajustado. O que aqui é chamado de ajuste são as instruções necessárias para que a *thread* executante não execute o segmento de código pertencente à *thread* que está sendo inicializada. Em outras palavras, a *thread* em estado *running* não deve chamar o método *run* do objeto que está executando o método *start*. Isto poderia ocorrer porque faz parte do método *start* a invocação do método *run*. Abaixo

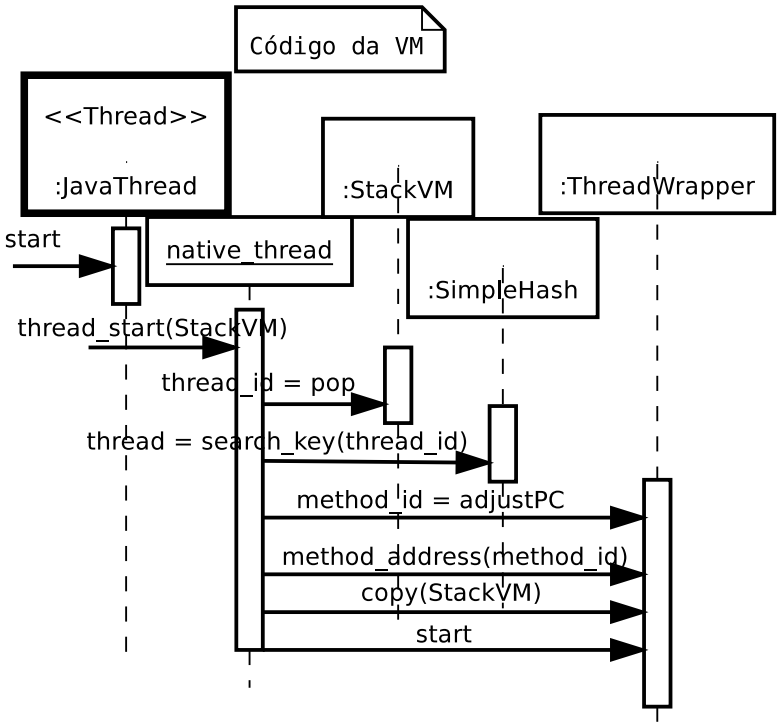

Figura 15 – Diagrama de sequência para a inicialização de uma *thread* JAVA

um trecho do código para exemplificar o que está sendo descrito:

```
public abstract class Thread {

  public Thread(){}

  public void start(){
        super.start(); //chamada do método nativo
        run();
  }

  public abstract void run();
}
```

Para isto, o PC da *thread* é incrementado em 4. Isto faz com que a esta não execute as instruções da máquina virtual necessárias para a invocação e retorno de um método. Consequentemente, após o incremento, o PC está apontando para o segmento de código que tem as instruções que representam exatamente a continuação do código, ou seja, o método *run* foi desconsiderado pela *thread* em execução. Por fim, o *adjustPC* retorna o identificador do método *run* que não foi executado. O identificador retornado é utilizado para que a *thread* que está sendo inicializada saiba qual o segmento de código que deve ser executado. Note que **skip** é o incremento do PC.

Em nível de sistema operacional, é neste momento que ocorre a primeira troca de contexto em relação a estas duas threads, a que está chamando o método *start* e a que vai executar o método *run*. Portanto a thread que invocou o método *start* pode entrar em estado suspenso e a *thread* inicializada executará o código pertencente ao método *run* da classe a qual ela pertence.

3.4 A PILHA

A implementação da NanoVM original foi realizada com uma pilha global, no entanto, segundo as especificações da JVM cada *thread* deve possuir sua própria pilha. Devido a este detalhe de decisão de projeto da NanoVM, fez-se necessário uma mudança na estrutura da máquina original.

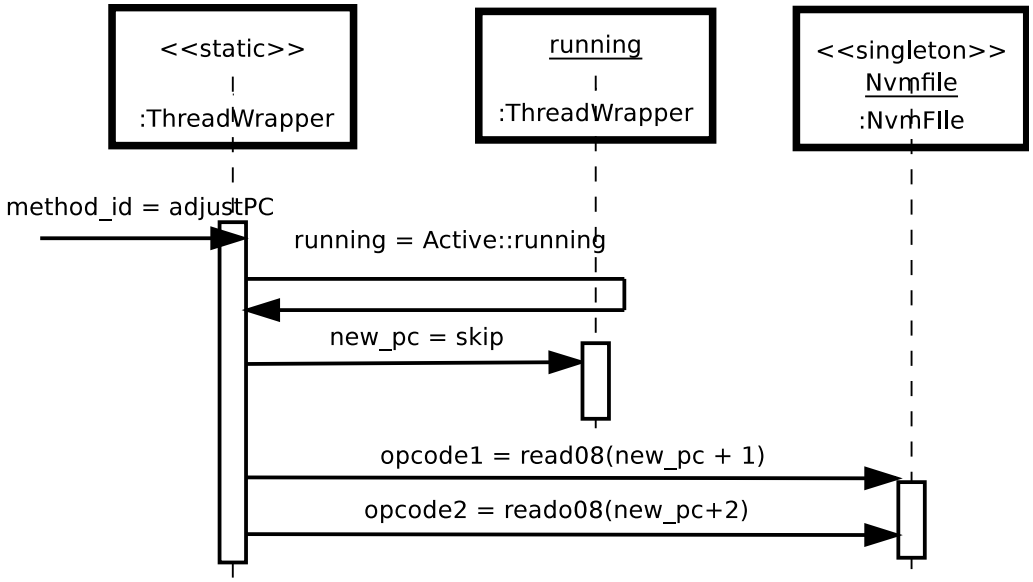

Figura 16 – Diagrama de sequência do método *adjustPC*

Podendo a *heap* e pilha serem unificadas, segundo a especificação da JVM, uma variável global apontando para um trecho da *heap* representava a pilha da NanoVM. No entanto, fez-se necessário criar a ideia de que cada *thread* possua sua própria pilha.

Na figura 17 pode ser visto o diagrama de sequência e as operações necessárias para que a *thread* que está sendo inicializada tenha acesso aos objetos alocados na *heap*. Assim, quando uma nova *thread* é construída, uma pilha também é criada. A criação de uma pilha nada mais é do que definir um endereço da *heap* como base desta nova pilha. Logo, cada objeto do sistema que representa uma pilha da NanoVM, tem como principal atributo a base da mesma. Para obtermos este endereço a seguinte operação é realizada:

$$basePilha \leftarrow baseDaUltimaPilhaCriada + tamanhoDaPilha \tag{3.1}$$

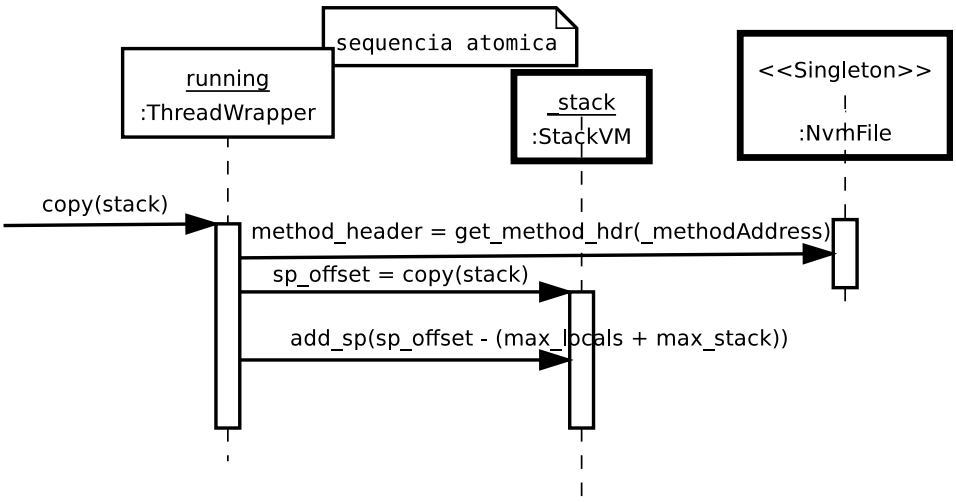

Figura 17 – Diagrama de sequência mostrando a criação de uma nova pilha

3.5 SINCRONIZAÇÃO

Para a sincronização das *threads* a especificação da máquina virtual JAVA determina a utilização de monitores. No entanto, nesta implementação foi utilizado o conceito de semáforos para realizar a

sincronização das tarefas, e não monitores.

Esta decisão teve como objetivo utilizar diretamente os mecanismos já implementados no sistema operacional EPOS. A classe JAVA que representa um semáforo é declarada como classe nativa, onde cada objeto instanciado é mapeado diretamente para um semáforo do sistema operacional, de forma análoga ao mapeamento realizado na criação das *threads* Logo, para cada semáforo criado no programa escrito em *Java* existe um semáforo do sistema operacional que o representa.

Com isto não é necessário implementar mecanismos de sincronização na máquina virtual, ou seja, o código da máquina virtual permanece enxuto, pois os mecanismos de sincronização de tarefas são realizados diretamente pelo sistema operacional.

4 AVALIAÇÃO DA INTEGRAÇÃO NANOVMEPOS

Esta capítulo apresenta os casos de estudos utilizados na verificação de corretude do suporte à concorrência, sincronização e desempenho da solução proposta. É também apresentada uma análise do código objeto de cada aplicação de teste, ou seja, é verificado o tamanho (*footprint*) do executável gerado para cada *benchmark* realizado. Os seguintes *benchmarks* foram realizados:

- Jantar dos Filósofos
- BubbleSort
- RingBenchMark

Estes aplicativos foram executados e compilados com as seguintes características:

- processador Atmega1281,
- velocidade do processador: 8MHZ,
- memória RAM: 8Kbytes,
- eeprom: 4Kbytes
- memória flash: 128Kbytes
- compilador: avr-g++ 4.0.2
- flags do compilador: -O2 -Wno-inline

A *flag* -O2 faz com que o compilador *gcc* realize a maior parte das otimizações possíveis sem *trade-off* de espaço e velocidade. O compilador não realiza desenrolamento de laços, nem substitui chamadas de funções por códigos *inline*. A *flag* *-Wno-inline* serve para impedir que a *flag* -O2 realize otimizações *inline*, as quais são feitas por padrão. Isto para impedir que o código executável ocupe mais espaço na memória em virtude das otimizações.

Vale lembrar que devido ao fato destes sistemas estarem executando apenas uma aplicação, ou seja, são sistemas dedicados, não existe variação de tempo entre diferentes execuções (considerando aplicações que não realizem operações de IO e que não tenha nenhuma interrupção de hardware habilitada), diferentemente do que pode ocorrer num *desktop* comum.

4.1 USO DE MEMÓRIA E DESEMPENHO

Para fins de avaliação de desempenho com uma única thread em execução foi usado o algoritmo de ordenação *bubble sort*. Como pode ser visto no algoritmo 1, o mesmo possui complexidade

$$O(n^2), \tag{4.1}$$

fazendo com que ele seja o suficiente para avaliar o desempenho do sistema e analisar o sistema sem que ocorra nenhuma interrupção externa. Portanto, é avaliado o *overhead* causado pela interpretação dos *bycodes*, já que a NanoVM interpreta *bytecodes*, e o ambiente de execução necessário para a execução da máquina virtual.

Algorithm 1 BubbleSort

```
for i in numbers do
  for j in numbers do
    notInOrder ←compare(i, j)
    if notInOrder then
      swap(N[i], N[j]
    end if
  end for
end for
```

Fooprint (bytes)					
Section	Nano	EPOS	Nano+EPOS	Nano_OO	Darjeeling
data	238	22	116	94	686
text	5596	15000	27766	14300	62202
bss	815	196	206	10	2105
total	7777	15218	29622	14404	78303

Tabela 1 – *Footprint do BubbleSort*

Desempenho (em segundos)				
NanoVM	Nano+EPOS	Darjeeling	NanoVM NanoVM + EPOS	Darjeeling NanoVM + EPOS
1.19s	1.96s	1.24	64.7%	58%

Tabela 2 – *BubbleSort: comparação de desempenho entre Darjeeling, NanoVM original e NanoVM integrada com o EPOS*

Na table 1 temos o *footprint* da aplicação *BubbleSort* no sistema operacional EPOS, da NanoVM integrada com o EPOS (*Nano+EPOS*), do código objeto da NanoVM integrada com o EPOS (*Nano_OO*), da NanoVM original (*Nano*) e ainda do código objeto da DARJEELING.

A tabela 1 sugere que a diferença de espaço ocupado na memória entre a NanoVM pura e a NanoVM_OO é devido ao acréscimo do suporte à concorrência. Entretanto, fazendo uma análise mais profunda no código gerado pelo compilador, utilizando o programa *avr-objdump*, foi descoberto que a principal razão do aumento no *footprint* do código foi a modificação realizada na pilha da máquina virtual. Com a utilização do programa *avr-objdump* foi possível analisar o código *assembly* gerado para a arquitetura AVR, e assim analisar a diferença entre o *assembly* gerado para a NanoVM não modificada, e a NanoVM_OO.

Independente das threads, o fato das pilhas serem objetos na nova implementação, com o objetivo de ajudar no suporte à *multi-thread* e seguir a especificação da máquina virtual JAVA, faz com que os métodos de manipulação das pilhas, *push* e *pop*, gerem um código de máquina diferente. Na versão original da NanoVM, estes métodos acessam uma variável global que representa a pilha, no entanto, estes métodos, na versão orientada a objeto, requerem um argumento que representa a pilha que está sendo manipulada. No contexto de programação orientada a objeto este parâmetro é passado automaticamente. Logo, o código gerado pelo compilador é equivalente ao código gerado por uma linguagem que não suporta o paradigma orientado a objeto onde o argumento necessário é fornecido explicitamente pelo programador. Ou seja, os métodos logo abaixo quando compilados pelo *gcc* resultam no mesmo código de máquina.

$$pop(stack)$$
$$stack.pop()$$

No entanto se analisarmos o método a seguir, onde a pilha é uma variável global, o código gerado é mais sucinto, pois a assinatura do método não possui parâmetros.

$$popStack()$$

Ora, analisando a implementação das 256 instruções da NanoVM notou-se que aproximadamente dois terços destas instruções utilizam duas chamadas de método de manipulação da pilha, e mais algum comando para manipular o valor retornado da pilha. Portanto, se para uma pilha global o compilador *gcc* gerava 3 instruções de máquina real para realizar uma instrução da máquina virtual, depois do suporte à múltiplas pilhas esta mesma instrução precisa destas 3 instruções mais as instruções adicionais de manipulação de argumentos de chamadas de funções, necessárias para realizar a chamada dos métodos da pilha da NanoVM. Isto explica a diferença de tamanho da NanoVM pura para a NanoVM integrada no EPOS. Além disso, se analisarmos a tabela 2 podemos notar esta relação na diferença de desempenho entre as máquinas, onde o desempenho da NanoVM integrada teve uma redução de aproximadamente 68%.

Na tabela 2 pode-se ver que o desempenho do protótipo é consideravelmente inferior ao da DARJEELING. Porém, a tabela 1 nos mostra que em termos de tamanho a NanoVM continua sendo mais compacta mesmo acrescida de suporte à multi-thread e mecanismos de sincronização.

Fooprint (byte) e desempenho(em micro-segundos		
Section	EPOS	Nano+EPOS
data	330	308
text	27286	29300
bss	200	206
total	27816	29814
spawn	570us	4600us
tempo/mensagem	50us	500us

Tabela 3 – Análise do Ring BenchMark

4.2 SINCRONIZAÇÃO

Com o objetivo de verificar a corretude da implementação do suporte a concorrência na NanoVM utilizando chamadas nativas do sistema operacional, foi utilizado o problema do jantar dos filósofos (DIJKSTRA, 1971). Para executar este programa foram necessárias 7 threads. Cinco destas threads representam os filósofos, uma delas é a responsável pela execução do ponto de entrada da aplicação JAVA, ou seja, o método *main* da aplicação JAVA. A thread faltante é a responsável por inicializar a JVM, ou seja, o ponto de entrada do sistema operacional. No caso do EPOS existe a *thread IDLE*, fazendo com que o sistema não seja totalmente desligado.

Após a comprovação do funcionamento dos mecanismos necessários para uma aplicação *muti-thread*, demonstrado pelo jantar dos filósofos, foi realizado o *RingBenchMark*(ERICSSON, 1998). Esta aplicação consiste em criar um anel de threads, após a criação destas uma mensagem é passada de thread para thread, simulando uma comunicação com topologia de anel. Assim é possível analisar o tempo necessário para que as threads se comuniquem entre si.

Analisando a tabela 3, é fácil notar que o otimizador de *bytecodes* consegue desempenhar sua função. Fazendo uma comparação entre as tabelas 1 e 3 nota-se que existe uma diferença significativa no tamanho do código gerado, na seção *data* para as duas aplicações escritas em C++. No entanto, se analisarmos o *footprint* do código gerado com a utilização da máquina virtual, ou seja, a aplicação escrita em JAVA, é possível perceber que o executável gerado tem uma diferença de menos de 3000 bytes. Estes quase 3000 bytes de diferença na seção *text* é devido a não utilização do mecanismo de sincronização (Semáforo) na aplicação *BubbleSort*.

Com isto em mente, pode-se dizer que se uma aplicação possui o *footprint* relativamente grande o *trade-off* entre espaço e desempenho pode ser uma boa opção. Isto porque os *bytecodes* otimizados pelo *nanovmTool* compensam o executável gerado para se ter o interpretador JAVA.

5 CONCLUSÃO

Esta monografia investiga a questão de como implementar uma máquina virtual orientada a aplicação, utilizando um sistema operacional que segue a metodologia ADESD. Com isso as ferramentas de uma *HLL VM* podem ajudar a indústria na produção de softwares para sistemas com alta restrição de memória, desempenho e energia, ou seja, sistemas embarcados.

Com fins de demonstrar a proposta, foi realizada a integração da NanoVM com o EPOS, sendo a NanoVM incrementada com mecanismos que dão suporte à programação concorrente. Tal integração mostrou a possibilidade de se ter uma máquina virtual multi-thread para sistemas com alta restrição de memória.

O consumo de memória gerado pelo ambiente de execução de uma máquina virtual foi reduzidoo mapeamento direto dos mecanismos já existentes no sistema operacional, como os necessários para que se possa ter múltiplas tarefas, incluindo troca de contexto e sincronização das mesmas.

Um dos maiores desafios deste trabalho foi de fato a implementação deste protótipo, entre os vários desafios destaca-se o fato da NanoVM não ser orientada a objetos, se soma a isso o fato desta máquina originalmente não ter sido projetada com suporte ao paralelismo, o que acabou implicando na modificação do núcleo da máquina.

Foram realizados experimentos para avaliar desempenho e consumo de memória dos programas executáveis. Sendo o principal foco deste trabalho o consumo de memória, o objetivo foi atingido, mantendo o custo de se ter uma máquina virtual multi-thread relativamente baixo quando comparado a outras máquinas virtuais.

Portanto, com o protótipo desenvolvido para este trabalho foi possível mostrar que com um sistema operacional orientado à aplicação, e com componentes bem estruturados pode se ter uma máquina virtual reduzida à um interpretador, bastando utilizar os componentes fornecidos pelo sistema operacional, para então criar o *runtime* da mesma. Neste sentido o coletor de lixo também deve ser um componente do EPOS, assim como a *heap* e pilhas das threads da VM. Todos estes componentes devem ser configuráveis no momento em que a aplicação alvo é determinada, podendo inclusive se optar por diferentes tipos de coletores de lixo, inclusive com coletores de lixos desenvolvidos para sistemas de tempo real, como o que foi desenvolvido para a máquina virtual Fiji.

Este trabalho demonstra que o tamanho do executável gerado para se ter uma máquina virtual JAVA pode ser consideravelmente pequeno se utilizarmos como base um sistema operacional orientado à aplicação, bem estruturado e que permita, na implementação de uma máquina virtual, a reutilização de componentes pertencentes ao mesmo. Vale lembrar que neste trabalho a maior parte do *overhead* causado no *footprint* do código objeto foi em virtude de uma decisão de projeto relacionado ás pilhas dá máquina virtual.

Para trabalhos futuros devem ser realizadas as integrações da HEAP e pilhas da máquina virtual com as que já são fornecidas pelo sistema operacional. Também deve ser adicionado um coletor de lixo ao sistema operacional.

REFERÊNCIAS BIBLIOGRÁFICAS

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. *Computer Networks*, v. 54, n. 15, p. 2787 – 2805, 2010. ISSN 1389-1286. <http://www.sciencedirect.com/science/article/pii/S1389128610001568>.

BØGHOLM, T.; HANSEN, R. R.; RAVN, A. P.; THOMSEN, B.; SØNDERGAARD, H. 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. <http://doi.acm.org/10.1145/1620405.1620427>.

BØGHOLM, T.; HANSEN, R. R.; RAVN, A. P.; THOMSEN, B.; SØNDERGAARD, H. 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. <http://doi.acm.org/10.1145/1850771.1850772>.

BROUWERS, N.; LANGENDOEN, K.; CORKE, P. Darjeeling, a feature-rich vm for the resource poor. In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. New York, NY, USA: ACM, 2009. (SenSys '09), p. 169–182. ISBN 978-1-60558-519-2. <http://doi.acm.org/10.1145/1644038.1644056>.

CRAIG, I. *Virtual Machines*. Springer, 2006. ISBN 9781852339692. <http://books.google.com.br/books?id=vIB-npHEbd4C>.

DIJKSTRA, E. W. Hierarchical ordering of sequential processes. *Acta Informatica*, Springer Berlin / Heidelberg, v. 1, p. 115–138, 1971. ISSN 0001-5903. 10.1007/BF00289519. <http://dx.doi.org/10.1007/BF00289519>.

ERICSSON. *Performance Measurements of Threads in Java and Processes in Erlang type @ONLINE*. jun. 1998. <http://www.sics.se/ joe/ericsson/du98024.html>.

FRÖHLICH, A. A. *Application-Oriented Operating Systems*. 2001.

GILOI, W. Konrad zuse's plankalk uuml;l: the first high-level, ldquo;non von neumann rdquo; programming language. *Annals of the History of Computing, IEEE*, v. 19, n. 2, p. 17 –24, apr-jun 1997. ISSN 1058-6180.

GOUGH, B. J.; STALLMAN, R. M. *An Introduction to GCC*. [S.l.]: Network Theory Ltd., 2004. ISBN 0954161793.

GROUP, O. *Operating system specification 2.2.3. Technical report type @ONLINE*. fev. 2005. <http://portal.osek-vdx.org/>.

HARBAUM, T. *NanoVM - Java for the AVR*. 2005. [http://www.harbaum.org/till/nanovm/index.shtml].

JAZELLE - ARM. [Online; accessed March 18, 2010]. <http://www.arm.com/products/processors/technologies/jazelle.php>. Acessado em 18 Mar. 2010.

LEVIS, P.; CULLER, D. MatÃ©: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 36, n. 5, p. 85–95, out. 2002. ISSN 0163-5980. <http://doi.acm.org/10.1145/635508.605407>.

LEWIS, J. A.; HENRY, S. M.; KAFURA, D. G.; SCHULMAN, R. S. An empirical study of the object-oriented paradigm and software reuse. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 26, n. 11, p. 184–196, nov. 1991. ISSN 0362-1340. <http://doi.acm.org/10.1145/118014.117969>.

LINDHOLM, T.; YELLIN, F. *Java Virtual Machine Specification*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201432943.

LUDWICH, M. K. *Método para Abstração de Componentes de Hardware para Sistemas Embarcados*. 131 p. Dissertação (Mestrado) — Federal University of Santa Catarina, Florianópolis, 2012. M.Sc. Thesis.

MICROSYSTEMS, S. *The K Virtual Machine @ONLINE*. jul. 2012. <http://java.sun.com/products/cldc/wp/>.

ORTIZ, C. E. *A Survey of Java ME @ONLINE*. jul. 2007. <http://developers.sun.com/mobility/getstart/articles/survey/>.

PAJIC, M.; MANGHARAM, R. Embedded virtual machines for robust wireless control and actuation. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. [S.l.: s.n.], 2010. p. 79 –88. ISSN 1080-1812.

PIZLO, F.; ZIAREK, L.; BLANTON, E.; MAJ, P.; VITEK, J. 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. <http://doi.acm.org/10.1145/1755913.1755922>.

POLPETA, F. V.; FRÖHLICH, A. A. Hardware mediators: a portability artifact for component-based systems. *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, Aizu, Japan, 2004. Pages 271-280.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, ACM, New York, NY, USA, v. 17, n. 7, p. 412–421, jul. 1974. ISSN 0001-0782. <http://doi.acm.org/10.1145/361011.361073>.

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.

SAMPSON, A.; DIETL, W.; FORTUNA, E.; GNANAPRAGASAM, D.; CEZE, L.; GROSSMAN, D. Enerj: approximate data types for safe and general low-power computation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 46, n. 6, p. 164–174, jun. 2011. ISSN 0362-1340. <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.

SMITH, J. E.; NAIR, R. The architecture of virtual machines. *Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 38, p. 32–38, 2005. ISSN 0018-9162.

THOMM, I.; STILKERICH, M.; WAWERSICH, C.; SCHRöDER-PREIKSCHAT, W. 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. <http://doi.acm.org/10.1145/1850771.1850788>.

TINYOS. *TinyOS type @ONLINE*. jul. 2012. <http://www.tinyos.net/>.

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. <http://doi.acm.org/10.1145/1531542.1531549>.

VITEK, J. Virtualizing real-time embedded systems with java. In: *Proceedings of the 48th Design Automation Conference*. New York, NY, USA: ACM, 2011. (DAC '11), p. 906–911. ISBN 978-1-4503-0636-2. <http://doi.acm.org/10.1145/2024724.2024926>.

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