Técnicas de otimização para o interpretador da K Virtual Machine

Randal Juliano Pereira Pinto

Dissertação de Mestrado Profissional

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

Instituto de Computação
Universidade Estadual de Campinas

Técnicas de otimização para o interpretador da K Virtual Machine

Randal Juliano Pereira Pinto

19 de novembro de 2003

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Sérgio Vanderlei Calvalcante
  Centro de Informática - UFPE
- Guido Costa Souza de Araújo
  Instituto de Computação - UNICAMP

UNIDADE IMECC
Nº CHAMADA
T/UNICAMP
P658t
V EX
TOMBO BC/59773
PROC. 16-117-04
C D
PREÇO 11,00
DATA
Nº CPD
BBID 322138

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Pinto, Randal Juliano Pereira

P658t Técnicas de otimização para o interpretador da K Virtual Machine / Randal Juliano Pereira Pinto -- Campinas, [S.P. :s.n.], 2003.

Orientador : Rodolfo Jardim de Azevedo.

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Interpretadores (Programas de computador). 2.VM/CMS (Sistema operacional de computador). 3. JAVA (Linguagem de programação de computador). I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Técnicas de otimização para o interpretador da K Virtual Machine

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Randal Juliano Pereira Pinto e aprovada pela Banca Examinadora.

Campinas, 19 de dezembro de 2003.

Rodolfo Jardim de Azevedo

Rodolfo Jardim de Azevedo (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

UNICAMP
BIBLIOTECA CENTRAL

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 17 de dezembro de 2003, pela Banca Examinadora composta pelos Professores Doutores:

Profº. Drº. Sergio Vanderlei Cavalcante
CIN-UFPE

Profº. Drº. Guido C. Souza Araújo
IC - UNICAMP

Profº. Drº. Rodolfo Jardim de Azevedo
IC - UNICAMP

© Randal Juliano Pereira Pinto, 2003.
Todos os direitos reservados.

Resumo

Dispositivos portáteis e personalizáveis têm se tornado ítens cada vez mais importantes em nossas vidas. Telefones celulares, organizadores pessoais, terminais de serviço possuem algumas características em comum como a funcionalidade limitada, restrições de tamanho e custo e, principalmente, são um mercado em plena expansão. Para atender a esta demanda, a Sun Microsystems Inc. extendeu o escopo da sua tecnologia Java com a introdução do Java 2 Platform Micro Edition (J2ME). Tendo em vista a característica limitada deste tipo de dispositivo, melhorias no desempenho da plataforma J2ME são essenciais para o seu desenvolvimento e aceitação. Diversas propostas de otimização são encontradas na literatura com enfoque principal nas tecnologias J2SE (Java 2 Standard Edition) e J2EE (Java 2 Enterprise Edition), porém quase nenhuma delas visa exclusivamente as necessidades da plataforma J2ME. Este trabalho propõe a aplicação de duas técnicas de otimização no interpretador da KVM. A primeira técnica baseia-se na redução da freqüência de acessos à memória e resultou em ganhos totais nos *benchmarks* analisados de 0,14% a 5,66% e ganhos em alguns testes de até 37,71%, com aumento no código fonte de apenas 0,62% em relação ao tamanho do executável original. A segunda baseia-se na redução de alguns passos do ciclo de interpretação de bytecodes e seu resultado geral variou de -0,90% até 5,32%, obtendo ganhos em alguns testes de até 18%, com aumento nos códigos fonte variando entre 0,0052% e 0,0142%.

Abstract

Mobile and customizable devices are increasingly becoming important items in our lives. Cellular phones, personal organizers, service terminals have many characteristics in common such as limited functionality, size and cost restrictions and specially are an expanding market. To attend this demanding market, Sun Microsystems Inc. extended the scope of the Java technology, introducing the Java 2 Platform Micro Edition (J2ME). Considering the limitations on this kind of device, performance improvements on the J2ME platform are essential to its development and general acceptance. Several optimization proposals can be found in the technical literature focused on J2SE (Java 2 Standard Edition) and J2EE (Java 2 Enterprise Edition) platforms, but almost none aims at the needs of the J2ME platform. The goal of this dissertation is to apply two optimization techniques to the KVM's interpreter. The first technique is based on reducing the number of memory accesses and resulted in a general performance increase from 0.14% to 5.66% and up to 37% for some benchmark programs. The increase on the source code size was only 0.62%, when compared to the original executable size. The second technique is based on reducing some steps of the bytecode interpretation cycle and its general results varied from a performance overhead of 0.90% to a performance increase of 5.32%, and up to 18% speedup for some of the benchmark programs. The increase in the source code size varied from 0.0052% to 0.0142%, when compared to the original executable size.

Agradecimentos

Gostaria de agradecer ao meu orientador, Rodolfo Jardim de Azevedo, por toda a dedicação, direcionamento técnico e empenho durante todo o desenvolvimento deste trabalho.

À minha família, por todo incentivo e educação que sempre recebi.

À Motorola, pelo tempo e apoio financeiro a mim concedidos para que este trabalho pudesse acontecer.

Ao Instituto de Computação, pela importante iniciativa em oferecer um curso de Mestrado Profissional, atendendo às necessidades de profissionais da área de pesquisa e desenvolvimento que buscam uma especialização sólida.

À minha família.

Sumário

Resumo vi

Abstract vii

Agradecimentos viii

1 Introdução 1
- 1.1 Organização 4

2 Trabalhos Relacionados 5
- 2.1 A Plataforma J2ME 5
- 2.2 Caracterização do Sistema 9
- 2.3 Técnicas de Otimização 11
  - 2.3.1 Máquinas Virtuais para Sistemas Tempo-Real 11
  - 2.3.2 O Garbage Collector 12
  - 2.3.3 Compiladores JIT (Just-in-Time) 13
  - 2.3.4 O Interpretador 14

3 Arquitetura e Caracterização da KVM 17
- 3.1 A Arquitetura da KVM 17
  - 3.1.1 O Mecanismo de Execução dos *bytecodes* 19
- 3.2 Caracterização da KVM 21
- 3.3 Oportunidades de Otimização 24

4 Técnicas de Otimização 27
- 4.1 Ambiente de Desenvolvimento e Parâmetros de Comparação 27
  - 4.1.1 Ambiente de Desenvolvimento 27
  - 4.1.2 Parâmetros de Comparação 28
- 4.2 Redução dos Acessos à Pilha 30
- 4.3 Criação de Super-Instruções 41

5 Conclusões e Trabalhos Futuros **49**
5.1 Trabalhos Futuros 50

A Resultados dos *Benchmarks* **53**
A.1 AggregateBenchCrypto 53
A.2 AggregateBenchDhrystoneHanoiSieve 53
A.3 AggregateBenchFireFractalPoleposition 54
A.4 AggregateBenchXMLParallel 54
A.5 JBenchmark 54
A.6 CLDCBench 54
A.7 MIDPBench 56

B Freqüências dos Pares de Bytecodes **57**

Bibliografia **60**

Lista de Tabelas

2.1 Classificação dos *bytecodes* quanto a sua funcionalidade 9

3.1 Número de *bytecodes* representando 90% e 80% dos *bytecodes* executados . 23
3.2 Proporção dos *bytecodes* do tipo Load em uma execução dinâmica 23
3.3 Caracterização da KVM quanto ao número de *pushes* e *pops* executados . . 24

4.1 Resultados gerais para a técnica de redução de acessos à pilha (RSA) . . . 38
4.2 Resultados para módulo MIDPBench - GraphicsFont 40
4.3 Resultados gerais para a técnica de criação de super-instruções (SI) 45
4.4 Resultados para o módulo AggregateBenchXMLParallel 46

B.1 AggregateBenchCrypto - Freqüência individual e dos pares de bytecodes . 57
B.2 AggregateBenchDhrystoneHanoiSieve - Freqüência individual e dos pares de bytecodes 57
B.3 AggregateBenchFireFractalPoleposition - Freqüência individual e dos pares de bytecodes 58
B.4 AggregateBenchXMLParallel - Freqüência individual e dos pares de bytecodes 58
B.5 JBenchmark - Freqüência individual e dos pares de bytecodes 58
B.6 CLDCBench - Freqüência individual e dos pares de bytecodes 59
B.7 MIDPBench - Freqüência individual e dos pares de bytecodes 59

Lista de Figuras

1.1 Projeção para o mercado de celulares até 2007 (fonte: Micro Java [25]) . . 3

2.1 A Plataforma J2ME (fonte: Datasheet J2ME [17]) 6

3.1 A Máquina Virtual Java (KVM) 18
3.2 Freqüência de ocorrência dos *bytecodes* na execução dos *benchmarks* 22
3.3 Freqüência de ocorrência dos pares de *bytecodes* na execução dos *benchmarks* 25

4.1 Implementação original da pilha para a KVM 31
4.2 Implementação modificada da pilha para a KVM 31
4.3 Máquina de estados para o cache de 1 elemento 32
4.4 Pilha - caso 1, estado FastInterpret_R0 34
4.5 Pilha - caso 1, estado FastInterpret_R1 35
4.6 Pilha - caso 2, estado FastInterpret_R0 36
4.7 Pilha - caso 2, estado FastInterpret_R1 37
4.8 Pilha - caso 3, estado FastInterpret_R0 38
4.9 Pilha - caso 3, estado FastInterpret_R1 39
4.10 Ciclo de execução de um *bytecode* 42
4.11 Ciclo de execução de uma super instrução - predição correta 43
4.12 Ciclo de execução de uma super instrução - predição errada 44

Capítulo 1

Introdução

Dispositivos portáteis e personalizáveis têm se tornado ítens cada vez mais importantes em nossas vidas. Telefones celulares, organizadores pessoais, terminais de serviço possuem algumas características em comum como funcionalidade limitada, restrições de tamanho e custo e, principalmente, formam um mercado em plena expansão. Porém, ao contrário de softwares para PCs, a indústria de dispositivos móveis não possui um sistema operacional ou plataforma dominante, isto faz com que desenvolvedores terceirizados tenham que aprender detalhes sobre a plataforma antes de escrever software para um determinado aparelho além de ter que adaptá-lo para cada fabricante de telefones e em muitos casos para diferentes telefones de um mesmo fabricante.

Essa diversidade de dispositivos e plataformas fez surgir no mercado a necessidade da criação de uma padrão que permitisse desenvolver aplicações comuns a vários dispositivos, de maneira a reduzir a complexidade e o custo de seu desenvolvimento.

Foi para atender a esta demanda que a Sun Microsystems Inc. extendeu o escopo da sua tecnologia Java com a introdução do Java 2 Platform, Micro Edition (J2ME), cuja proposta é a de proporcionar um ambiente multi-plataformas, de conteúdo dinâmico, seguro e que já possui uma grande comunidade de desenvolvedores.

O interesse no potencial da tecnologia Java como plataforma para distribuição de jogos e outros conteúdos para dispositivos sem-fio vem aumentando muito nos últimos 2 anos, a projeção de mercado é de que em 4 anos o número de celulares usando Java aumente aproximadamente 10 vezes, representando 74% do total desses aparelhos, comparados com os atuais 11%, como apresentado na Figura 1.1 (mais detalhes em [25]).

Tendo em vista a característica limitada deste tipo de dispositivo, melhorias no de-

sempenho da plataforma J2ME são essenciais para o seu desenvolvimento e aceitação. Diversas propostas de otimização são encontradas na literatura com enfoque principal nas tecnologias J2SE (Java 2 Standard Edition) e J2EE (Java 2 Enterprise Edition), porém quase nenhuma delas visa exclusivamente as necessidades da plataforma J2ME.

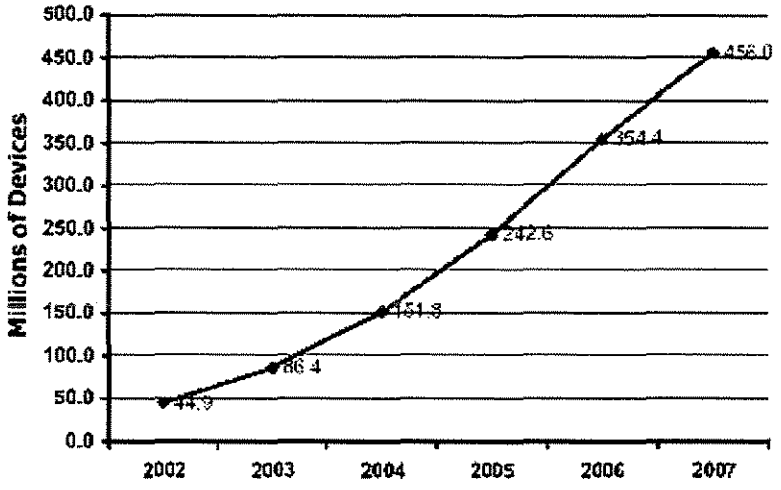
O objeto de estudo deste trabalho dentre as várias camadas da plataforma J2ME é a máquina virtual Java. Dentro deste tópico, são encontrados trabalhos de otimização na área do interpretador, do *garbage collector*, compiladores JIT (*Just-in-Time*) em substituição ao interpretador, e técnicas para aplicação desta em sistemas tempo-real.

Depois de alguma pesquisa na literatura focamos nosso trabalho na implementação da KVM através de um interpretador, pois esta é a técnica que menos demanda recursos específicos do hardware para ser implementada e por este motivo é a mais utilizada na implementação de KVMs. A arquitetura do interpretador é baseada numa máquina de pilha e apresenta algumas oportunidades de melhoria principalmente no que se refere a acessos à memória, que na maioria dos sistemas de computação é muito custosa. Outra oportunidade de melhoria está na freqüência de utilização dos *bytecodes* onde, assim como acontece entre compiladores e processadores, um conjunto pequeno de instruções é utilizado com maior freqüência, aumentando também a freqüência de determinados pares de instruções.

Em resumo, este trabalho propõe a aplicação de duas técnicas de otimização no interpretador da KVM, a primeira técnica baseia-se na redução da freqüência de acessos à memória através da passagem de alguns parâmetros por registrador, o que resultou em ganhos totais de 0,14% a 5,66% e ganhos em alguns testes de até 37,71%, nos *benchmarks* analisados, com aumento no código fonte de apenas 0,62% em relação ao tamanho do executável original. A segunda baseia-se na redução de alguns passos do ciclo de interpretação de *bytecodes* através da criação de super-instruções e seu resultado geral variou de -0,90% até 5,32% em um *benchmark* específico, obtendo ganhos em alguns testes de até 18%, com aumento nos códigos fonte variando entre 0,0052% e 0,0142%.

Wireless Java Market Projections

Global Shipments Java Handsets (2002 - 2007)



	2002	2003	2004	2005	2006	2007
Total Handsets (Mil.)	401.8	450.8	509.1	550.4	584.7	613.7
Java Handsets (Mil.)	44.9	86.4	151.8	242.6	354.4	456.0
Java Handset Percent	11%	19%	30%	44%	61%	74%

Source: Zelos Group

Figura 1.1: Projeção para o mercado de celulares até 2007 (fonte: Micro Java [25])

1.1 Organização

Este trabalho está organizado da seguinte forma: No Capítulo 2, os trabalhos relacionados são mostrados, indicando as vantagens e desvantagens de cada um deles. No Capítulo 3 é apresentada uma visão geral da arquitetura da máquina virtual e também uma caracterização do interpretador da KVM quanto à freqüência de seus *bytecodes* e utilização da pilha, compondo a motivação das técnicas abordadas neste trabalho. No Capítulo 4 são apresentadas as duas técnicas de otimização implementadas neste trabalho bem como seus resultados e possíveis aplicações. Finalmente as conclusões e possíveis extensões deste trabalho estão descritas no Capítulo 5.

Capítulo 2

Trabalhos Relacionados

Assim como em *hardware* há uma constante busca pelo aumento da performance dos processadores, o estudo de técnicas para a melhoria do desempenho de máquinas virtuais é alvo de constantes pesquisas em *software*. Especificamente para a KVM, nosso alvo de estudo, melhorias na performance são bastante significativas visto que esta é projetada para sistemas limitados em recursos.

Não foram encontrados, na literatura pesquisada, trabalhos que abordassem especificamente a implementação da máquina virtual para a plataforma J2ME. Utilizamos então, como literatura de referência, trabalhos desenvolvidos para a máquina virtual Java para desktop, a JVM. Na Seção 2.1 deste capítulo é descrita a arquitetura da plataforma J2ME, na Seção 2.2 são apresentados trabalhos de caracterização da máquina virtual Java e na Seção 2.3 são descritos os métodos de otimização para a JVM encontrados na literatura.

2.1 A Plataforma J2ME

Nesta seção será descrita a plataforma estudada neste trabalho, a Java 2 Platform Micro Edition (J2ME) [13, 17]. A J2ME faz parte da plataforma Java 2 criada pela Sun Microsystems Inc. e tem como alvo dispositivos dedicados e móveis.

Com o objetivo de proporcionar uma distribuição flexível e customizável a arquitetura J2ME foi projetada para ser modular e escalável através de camadas de software construídas sobre o sistema operacional do sistema hospedeiro. Estas camadas são ilustradas na Figura 2.1 e descritas a seguir.

Máquina Virtual Java: A Java Virtual Machine (JVM) é a base fundamental das pla-

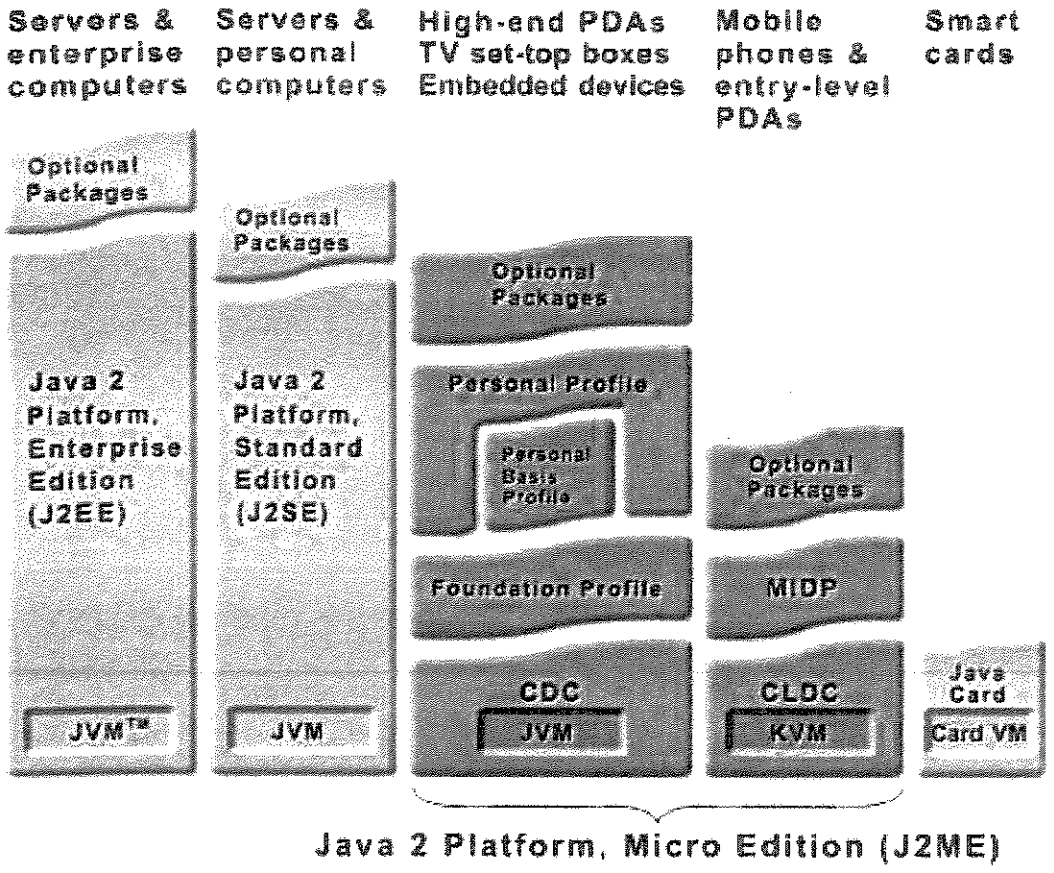


Figura 2.1: A Plataforma J2ME (fonte: Datasheet J2ME [17])

taformas Java e Java 2, sendo o componente da tecnologia responsável pela sua independência de hardware e sistema operacional. Ela é uma máquina computacional abstrata que, como uma máquina real, tem um conjunto de instruções e manipula várias áreas de memória em tempo de execução. Ela não interpreta diretamente códigos escritos em Java mas sim um formato binário particular, o formato de arquivo *class*. Um arquivo *class* contém instruções da JVM (ou *bytecodes*) e uma tabela de símbolos, além de outras informações derivadas. Por segurança, a JVM impõe um formato rígido e restrições estruturais ao código do arquivo *class*, no entanto qualquer linguagem com uma funcionalidade que pode ser expressa em

termos de um arquivo *class* pode ser hospedada pela JVM.

K Virtual Machine: A K Virtual Machine (KVM) é uma JVM compacta e portável especificamente projetada para dispositivos com limitações de recursos. O objetivo do projeto da KVM foi criar a menor JVM possível que mantivesse todos os aspectos centrais da linguagem de programação Java mas que rodasse em um dispositivo com recursos limitados, usando apenas algumas centenas de kilobytes de memória. O "K"em KVM significa "kilo", ela recebeu este nome pois seus requisitos de memória são medidos em kilobytes enquanto em sistemas para desktop mede-se em megabytes. A KVM é projetada para microprocessadores de 16/32 bits com arquitetura RISC ou CISC, o que tipicamente é a configuração de telefones celulares digitais, pagers, organizadores pessoais e pequenos terminais de venda. O requisito mínimo de memória para uma implementação da KVM é em torno de 129kB, incluindo a virtual machine, o mínimo de bibliotecas de classes Java, especificado pela configuração, e algum espaço no *heap* para rodar as aplicações. O papel atual de uma KVM nos diversos dispositivos pode variar significativamente. Em algumas implementações a KVM é usada sobre uma pilha de software já existente para dar ao dispositivo a habilidade de carregar e rodar dinamicamente conteúdos Java interativos e seguros. Em outras implementações a KVM é usada no "baixo nível" para também implementar o software de sistema e aplicações do dispositivo na linguagem Java.

Camada de Configuração: é a menos visível para o usuário porém é a base para a construção da camada de perfil, definindo um mínimo de funcionalidades e bibliotecas disponíveis numa determinada categoria de dispositivos. Atualmente há dois tipos disponíveis de configuração para a J2ME: Connected Limited Device Configuration (CLDC) e Connected Device Configuration(CDC).

CLDC: É a menor das duas configurações e foi desenvolvida para dispositivos com conexões de rede intermitentes, processadores mais lentos e com limitações de memória, como por exemplo telefones celulares e PDAs.

CDC: Esta configuração foi projetada para dispositivos com maior disponibilidade

de recursos como receptores de TV a cabo e gateways residenciais. O CDC inclui uma máquina virtual Java completa e um sub-conjunto muito maior da plataforma Java 2 Standard Edition (J2SE).

Camada de Perfil: é a mais visível para os desenvolvedores de aplicação e define um conjunto mínimo de *Application Programming Interfaces* (APIs) disponíveis numa determinada categoria de dispositivos e que definem o ciclo de vida de uma aplicação, interface com o usuário e acesso a propriedades específicas do dispositivo. Aplicações escritas para determinado perfil são portáveis para qualquer dispositivo que suporte aquele perfil. As configurações disponíveis atualmente para J2ME são: *Mobile Information Device Profile* (MIDP), *Foundation Profile* (FP), *Personal Profile* (PP) e *Personal Basis Profile* (PBP) além de alguns pacotes opcionais (*Optional Packages*).

MIDP: Este perfil oferece as funcionalidades básicas para uma aplicação móvel, entre elas estão: interface com o usuário, conexão de rede, repositório de dados e gerenciamento de aplicações. O MIDP, combinado com a CLDC constitui um ambiente de execução Java completo, visando minimizar os consumos de memória e energia.

FP: Os perfis para a CDC são organizados em camadas de forma a poderem ser adicionados conforme as necessidades da aplicação nos diferentes tipos de dispositivos. O FP pode usado por aplicações embarcadas, sem interface com o usuário e que acessem a rede, como gateways residenciais. É o perfil de mais baixo nível para a CDC.

PP: É o perfil da CDC para dispositivos que necessitem de uma interface com usuário. Ele inclui o conjunto completo de bibliotecas do *Java Abstract Window Toolkit* (AWT) além de compatibilidade com aplicações para a Web. Exemplos de dispositivos que usam o PP são: consoles para jogos e PDAs mais avançados.

PBP: É um sub-conjunto do PP e constitui um conjunto básico de APIs para aplicações que acessam a rede e precisam de um nível básico de representação gráfica ou que usam uma biblioteca gráfica especializada. Exemplos de dispositivos que usam o PBP são: sistemas de telemática, terminais de informação

e receptores de TV a cabo.

Pacotes opcionais: Estes pacotes são criados para atender à requisitos específicos do mercado como, por exemplo, bluetooth, mensagens de texto, multimídia, além de requisitos específicos do fabricante do dispositivo.

Apresentamos nesta seção uma descrição da plataforma J2ME, nas seções 2.2 e 2.3 apresentaremos os trabalhos relacionados com a proposta deste trabalho e no Capítulo 3 detalharemos as características da KVM, que é o alvo desse estudo.

2.2 Caracterização do Sistema

Radhakrishnan em [29] faz uma caracterização bastante detalhada dos sistemas de execução Java, analisando diversas implicações arquiteturais sob perspectivas de hardware e da implementação da JVM, num ambiente rodando o *benchmark* SPECjvm98 em duas implementações diferentes da JVM: Sun JDK 1.1.6 e Kaffe VM 0.9.2. Os *benchmarks* foram rodados via linha de comando e não usaram nenhum serviço de gráficos ou de rede.

Com relação às características observadas no nível dos *bytecodes*, é proposta uma classificação para estes baseada em suas funcionalidades inerentes (Tabela 2.1).

Bytecodes	Descrição	Exemplos
Load	Carrega dados de variáveis locais na pilha	iload0
Store	Carrega dados da pilha em variáveis locais	istore1
Stack	Push e pop de operandos além de duplicação e swap de dados na pilha	bipush dup
Constant Pool	Alocação de elementos no pool de constantes. A dealocação é feita pelo garbage collector	new getfield
ALU	Instruções lógicas e aritméticas	add, lshr
Branch	Testa por uma condição e muda o contador de programa em função disto.	ificmplt
Jump	Incrementos e decrementos não unitários do contador	goto
Method Call	Difere dos Branches e dos Jumps pois é necessário alocar um novo frame para guardar a pilha do método e dealocá-lo para os returns.	invokestatic return

Tabela 2.1: Classificação dos *bytecodes* quanto a sua funcionalidade

Os resultados obtidos para o *benchmark* SPECjvm98 mostram que 35,5% dos *bytecodes* executados pertencem ao grupo Load, ou seja, a transferência de elementos entre a memória alocada para variáveis locais e a pilha domina, o que é uma conseqüência da arquitetura desta máquina virtual, que é uma máquina de pilha (veja mais detalhes sobre a arquitetura da KVM na Seção 3.1). Conseqüentemente é esperado que técnicas como a combinação de instruções, proposta inicialmente para compiladores JIT, possam melhorar o desempenho geral da VM.

Outra característica observada foi o tamanho dinâmico dos métodos (em número de bytecodes executados), uma vez que invocar métodos Java é bastante custoso pois envolve a alocação de um ambiente de execução e uma nova pilha para cada novo método. Foi observada uma distribuição trinodal, onde a maioria dos métodos possuem 1, 9 ou 26 *bytecodes*, o que, segundo os autores, pode ser uma característica do ambiente de execução (uso freqüente de determinadas bibliotecas) e não um comportamento comum.

Uma outra análise mostra que alguns poucos *bytecodes* constituem a maioria dos *bytecodes* executados. Na maioria dos *benchmarks*, menos de 45 *bytecodes* representam 90% do total de *bytecodes* executados e menos de 33 representam 80% deste total, numa máquina que possui em torno de 200 bytecodes distintos.

O'Donoghue em [28] faz uma análise da ocorrência de pares de *bytecodes* numa seqüência dinâmica para um conjunto de *benchmarks* comerciais da JVM. Por esta análise nota-se que um número muito pequeno de pares de *bytecodes* são utilizados com uma freqüência significativamente maior e, naquele ambiente, notoriamente o par ALOAD_0 seguido de GETFIELD. Outro padrão observado foi que a freqüência de uso dos pares obedeceu a lei das potências, com decréscimo logarítmico ao longo dos pares menos freqüentes.

Em [11] Horgan propõe quatro níveis em que um programa Java pode ser analisado:

- Estaticamente, no nível do código fonte: estudos neste nível são similares aos feitos em programas escritos em outras linguagens, onde métricas de software podem ser aplicadas.

- Estaticamente, no nível dos *bytecodes*: neste nível o uso dos *bytecodes* pode ser analisado com propósitos de otimização e compressão.

- Dinamicamente, no nível dos *bytecodes* de uma maneira independente: esta informação pode ser útil para determinar potenciais otimizações ou para estimar a

cobertura e efetividade de programas comumente usados para *benchmarking*.

- Dinamicamente, numa VM e arquitetura específica: esta é a base de estudos para otimização de performance e também para comparação de performance entre VMs.

No caso específico deste trabalho concentraremos nossa análise no último nível pois nosso interesse é caracterizar uma VM específica e após aplicar algumas técnicas de otimização, compará-la com a implementação original.

2.3 Técnicas de Otimização

Com base nas características da máquina virtual e da sua aplicação diversas técnicas de otimização são encontradas na literatura. Nesta seção serão apresentadas técnicas de otimização para a JVM com enfoque nas seguintes áreas: máquinas virtuais para sistemas tempo-real, *garbage collector*, compiladores JIT e interpretador.

2.3.1 Máquinas Virtuais para Sistemas Tempo-Real

Sistemas tempo-real também podem fazer uso de uma máquina virtual Java como ambiente de execução de programas, no entanto algumas modificações e extensões precisam ser feitas na plataforma de maneira a satisfazer os requisitos de um sistema de controle tempo-real. As principais barreiras da arquitetura da plataforma Java para seu uso em sistemas tempo-real são a falta de uma semântica de escalonamento de *threads*, a falta de controle sobre o escalonamento do *garbage collector* que pode aleatoriamente bloquear a execução de todas as *threads* dentro da máquina virtual, a falta de um mecanismo eficiente para o tratamento de eventos assíncronos (característica principal de um sistema tempo-real) e a falta de uma API de acesso à um relógio com resolução de tempo-real.

Trabalhos visando a extensão e otimização da máquina virtual Java para sistemas tempo-real podem ser encontrados em Chiao [1] e Lee [22]. O trabalho de Chiao [1] propõe uma extensão da JVM para resolver os problemas de tratamento de eventos assíncronos e acesso à um relógio com resolução de tempo-real através da extensão desta com uma biblioteca para tempo-real. Lee [22] se preocupa mais com sistemas com severa restrição de memória e propõe uma técnica de compressão de *bytecodes* que pode reduzir o código fonte em até 30%.

O enfoque do nosso trabalho porém, é manter a portabilidade da implementação de referência e também dos programas executados nesta máquina virtual. Os trabalhos de Chiao [1] e Lee [22] eliminam a portabilidade de suas implementações, seja pela extensão das bibliotecas Java ou pela implementação do interpretador em código nativo (assembler) da máquina destino, portanto não atendem aos nossos propósitos.

2.3.2 O Garbage Collector

Os problemas de performance do *garbage collector (GC)* têm sido estudados por mais de três décadas e este é uma das principais barreiras para o uso da plataforma Java em sistemas de tempo real. Existem muitos métodos propostos para melhorar a performance do GC, como melhores *layouts* para a *heap*, algoritmos adaptativos e mesmo uma implementação mais eficiente para a implementação de referência. Na sua última implementação de referência, a Java HotSpot Virtual Machine, a Sun deu especial atenção à este problema, propondo um esquema para o GC que reduz em até 80% os tempos de pausa, segundo observação de Dykstra em [4].

As técnicas e algoritmos para a otimização do GC baseiam-se principalmente no estudo do ciclo de vida dos objetos. Nesta área podemos citar os estudos desenvolvidos por Yang [34] e Dykstra [4] e o GC para sistemas tempo-real proposto por Siebert [31]. Yang [34] propõe uma abordagem para o GC não só através do espaço total ocupado pelos objetos mas também pelo tempo de vida de cada um destes objetos. Segundo seus resultados experimentais, através desta abordagem, o número de chamadas ao GC pode ser diminuído significativamente e mantendo quase o mesmo volume de *heap* da abordagem puramente espacial. Dykstra [4] faz um estudo comparativo entre a performance do GC na implementação original da JVM e a HotSpot Virtual Machine, destacando o impacto de cada uma das técnicas empregadas nesta última. Siebert [31] propõe um mecanismo de GC baseado na modificação da implementação do sistema de *threading*, do mecanismo de busca de referências e no layout dos objetos de forma a prover garantias de tempo-real ao sistema e de uso irrestrito da linguagem.

Apesar de ser uma área bastante estudada o tempo de execução total do GC em comparação ao tempo total de execução do interpretador é significativamente menor e varia de 0,69% a 28,8% contra 56,06% a 97,99% do interpretador no *benchmark* SPECjvm98, conforme observado no trabalho de Yang [34]. Como o intuito deste trabalho é a otimização

do tempo total de execução de um programa Java decidimos por concentrar nosso estudo no interpretador.

2.3.3 Compiladores JIT (Just-in-Time)

Compiladores convencionais são projetados para produzir códigos altamente otimizados sem prestar muita atenção ao tempo de compilação. Com o desenvolvimento de compiladores dinâmicos para a linguagem Java, o objetivo é que sua performance se aproxime de uma execução de códigos binários gerados em C/C++, por exemplo.

O objetivo do projeto de um compilador JIT é o de gerar um código rápido no menor tempo de compilação possível. Krall [21] propõe um algoritmo rápido para a tradução de *bytecodes* em código de máquina para processadores RISC, este algoritmo realiza combinação de instruções, eliminação de cópias e alocação de registradores. Os resultados desta implementação demonstram uma redução de sete vezes no tempo de compilação além de um código de máquina ligeiramente mais rápido. Newhall [27] comprova a eficácia dos compiladores JIT através da comparação com uma máquina virtual interpretada: o aumento de velocidade pode chegar até a dezenove vezes. Uma proposta ainda mais eficaz é apresentada pela Sun em sua máquina virtual HotSpot [16] onde o código é inicialmente rodado num interpretador e analisado durante sua execução para que sejam identificados os *"hot spots"*, ou trechos mais executados do programa e só então estes métodos são compilados. Uma técnica bastante similar à proposta na máquina virtual HotSpot é apresentada por Weimin [33] porém, ao invés de identificar trechos de código freqüentemente executados, esta técnica propõe a compilação de métodos em paralelo com a interpretação de alguns trechos de código, o que evita a parada da máquina virtual para aguardar a compilação de algum método.

Notoriamente uma máquina virtual que usa um compilador JIT é superior a uma puramente interpretada, porém um compilador deste tipo demanda um espaço em memória e uma capacidade de processamento considerável, não sendo assim aplicável à maioria dos sistemas embarcados, cuja característica é a restrição de recursos.

2.3.4 O Interpretador

Compiladores JIT proporcionam uma performance superior mas possuem a desvantagem de terem que ser substancialmente reescritos para cada arquitetura para a qual estes são portados além de demandar muita memória e capacidade de processamento. A vantagem do interpretador é a facilidade de desenvolvimento e manutenção, a necessidade reduzida de memória, além de poder ser portado para novas arquiteturas com um mínimo de modificações.

Existem várias abordagens para a melhoria na performance de um interpretador, algumas delas se baseiam no uso de técnicas desenvolvidas para compiladores e adaptadas para interpretadores e outras aproveitam-se de características da arquitetura do computador em que vão ser hospedados.

Na linha de aplicação de técnicas de compiladores ao interpretador, podemos citar os trabalhos de Ertl [5], Maierhofer [23, 24], Gregg [9, 10], VanDrunnen [32], Koopman [20] e Prabhas [2]. Uma descrição de cada uma destas contribuições é apresentada a seguir.

Em sua tese de doutorado Ertl [5] propõe uma técnica de otimização para a JVM através da eliminação de acessos à memória para os ítens do topo da pilha mais acessados e também pela combinação das atualizações do ponteiro de pilha. Esta técnica consiste em criar uma máquina de estados para o interpretador onde, dependendo do número de elementos da pilha armazenados em registradores, os *bytecodes* têm uma implementação diferente. São apresentadas também algumas técnicas de organização destes estados, tratamento de *overflow* e *underflow* da *cache* e de gerenciamento de múltiplas pilhas. Não foram apresentados resultados detalhados desta implementação, apenas mostrou-se que fazer o *cache* em registrador de um elemento da pilha resultou em ganhos de 11% e 7% em dois *benchmarks* distintos em conseqüência de uma redução de 8% e 5% respectivamente nos acessos à pilha.

Maierhofer em [23] e [24] apresenta um algoritmo para a construção de programas ótimos, para um conjunto limitado de expressões. Este algoritmo baseia-se na análise de blocos de operações onde uma variável é usada mais de uma vez e o estado inicial e final da pilha seja vazio, propondo então uma organização desses *bytecodes* para minimizar o número de acessos à pilha. Esta proposta atua alterando a seqüência de *bytecodes* gerados, funcionando como um pós-compilador.

Em [20] Koopman conduziu um estudo para a otimização de códigos que usam a pilha através da substituição de *loads* e *stores* por manipulações da *stack*. Para isto, ele se baseou no simples processo de procurar por pares de *loads* de uma variável local num bloco de código e preservar o valor do primeiro *load* para que o segundo pudesse ser eliminado, processo o qual ele chamou de *stack scheduling*.

Em [32] também é apresentada uma família de transformações locais em *bytecodes* que manipulam a pilha. Cada transformação elimina uma instrução que carrega o conteúdo de uma variável local e substitui por um código que manipula diretamente a pilha. Se todos os carregamentos desta variável puderem ser substituídas por manipulações diretas da pilha, o *store* desta variável pode ser eliminado e conseqüentemente a própria variável.

Gregg em [9] e [10] apresenta uma alternativa mais eficiente ao interpretador baseado em *"switch"*, através do uso de *direct threading*[1]. Nesta técnica, instruções são representadas pelo endereço da rotina que a implementa e o seu *dispatch* consiste em buscar aquele endereço e fazer o *branch* para a rotina. Esta técnica não pode ser implementada em ANSI C pois este não possui labels de primeira classe[2].

Em [2] Prabhas propõe uma análise *bottom-up* para a otimização da JVM. Esta análise baseia-se no reconhecimento de seqüências de *bytecodes* durante a execução de programas de *benchmark* e na substituição destas seqüências por versões especializadas dos *bytecodes* como uma extensão à máquina virtual original. Esta é uma alternativa independente de plataforma aos compiladores Just-in-Time, que também pode ser usada em implementações da VM para sistemas com restrições de recursos, pois o *overhead* causado pela geração de *bytecodes* especializados pode ser controlado, através da limitação do número destes, de acordo com a capacidade do sistema.

Na linha de propor técnicas de otimização baseadas nas características de computadores modernos podemos citar os trabalhos de Ertl [6, 7], Kim [19] e Hsieh [12].

Ertl em [6] faz um estudo sobre a eficiência de esquemas de predição de *branches* em vários interpretadores, a conclusão deste trabalho é de que para interpretadores baseados em *direct threading* [9, 10], o número de erros nesta predição é muito menor e a velocidade é aumentada em duas vezes se comparado a interpretadores baseados em *"switch"*. Estes

[1]Técnica de implementação de interpretadores para máquinas virtuais, introduzida por J.R. Bell em 1973, onde cada *bytecode* no conjunto de instruções é o endereço de um código para executar a operação requisitada

[2]Mecanismo usado em *direct threading* para permitir que seja feito o *jump* para um ponteiro de função

dados são confirmados no estudo conduzido por Hsieh [12] que faz um estudo do comportamento da cache e da eficiência na predição de *branches* para interpretadores baseados em *"switch"*, o resultado deste estudo é a de que esta é uma área que pode ser bastante explorada de forma a otimizar interpretadores para a arquitetura de computadores modernos.

Em [7], Ertl apresenta algumas técnicas de predição de *branches* de forma a minimizar o número de predições erradas. Estas técnicas levam em consideração que o processador utilizado possui uma memória *cache* e esta usa um mecanismo de predição do tipo BTB (*Branch Target Buffer*) [3]. Estas técnicas se baseiam no uso de super-instruções e replicação de instruções, e se baseiam no fato de que quanto menor a freqüência de determinada instrução maior a probabilidade do BTB de acertar a predição do *branch*. No caso da replicação de instruções, cada instrução replicada passa a possuir um *branch* diferente de *dispatch* portanto, para o mecanismo do BTB, cada uma representa uma instrução diferente, melhorando sua performance. Para seleção das instruções a serem replicadas é utilizado um algoritmo round-robin. Para a criação das super-instruções é aplicado um algoritmo guloso sobre a freqüência das instruções.

Kim [19] propõe uma nova arquitetura de processadores de forma a beneficiar o paralelismo de *bytecodes* Java, além de um novo mecanismo para a implementação do paralelismo de instruções. Os resultados desta proposta são ganhos de 68% à 77% em velocidade de processamento.

Finalmente Ertl em [8] combina diversas técnicas de otimização de interpretadores para a construção de um gerador de interpretadores eficientes. Como entrada deste gerador é usado um conjunto de características e dados de uso da *stack*, super-instruções, tipos de dados e a partir destas entradas este programa se propõe a gerar o melhor código possível fazendo o melhor uso das técnicas de otimização por ele conhecidas. Os resultados são analisados considerando cada uma das técnicas de otimização aplicadas. Notam-se resultados muito bons para a técnica de fazer *cache* em registrador do topo da pilha e também para a criação de super-instruções e técnicas de predição de *branches*.

Como não foram encontrados na literatura trabalhos que abordassem especificamente a KVM, este trabalho é baseado em técnicas desenvolvidas para a JVM. No próximo capítulo serão apresentadas as arquiteturas da KVM e do seu interpretador e também as oportunidades de otimização exploradas neste trabalho.

Capítulo 3

Arquitetura e Caracterização da KVM

Neste capítulo é feita uma breve descrição da arquitetura da KVM detalhando em especial seu interpretador. É feita também uma caracterização dinâmica desta quanto à freqüência de ocorrência de *bytecodes* e acessos à pilha. Finalmente são apresentados os dados que motivaram as implementações deste trabalho.

3.1 A Arquitetura da KVM

Nesta seção é feita uma descrição de cada um dos componentes que compõem a arquitetura da KVM. A Figura 3.1 mostra a arquitetura da KVM e seus componentes são descritos a seguir:

RAM e ROM: a memória ROM contém informações sobre as classes, o código da máquina virtual e os métodos nativos. A memória RAM é dividida em *heap*, pilha Java e pilha de métodos nativos.

Interpretador : é o centro da KVM e é responsável pela decodificação, interpretação e execução dos *bytecodes.*

JNI (Java Native Interface): é uma interface de programação nativa que possibilita que métodos nativos trabalhem com qualquer implementação da plataforma Java para um computador específico. Ela é a principal responsável pela independência

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

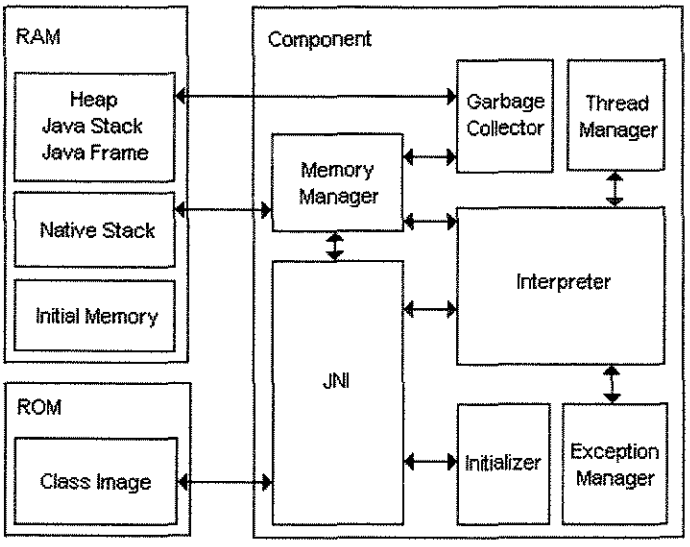


Figura 3.1: A Máquina Virtual Java (KVM)

de plataforma dos programas Java e também pode ser extendida para lidar com situações que não podem ser resolvidas por programas puramente Java.

Memory Manager: o gerenciador de memória controla a área do *heap* na memória RAM, ele informa ao *garbage collector* o estado da memória para as estruturas de dados.

Exception Manager: a especificação da linguagem Java define várias classes de exceção, o gerenciador de exceções é o responsável por observar erros e exceções e repassá-los adequadamente.

Thread Manager: é o responsável pelo escalonamento de *threads*, este gerenciamento depende da implementação e pode ser feito pelo sistema operacional ou dentro da própria KVM.

Garbage Collector: é o responsável por coletar e liberar áreas de memória sem uso.

3.1.1 O Mecanismo de Execução dos *bytecodes*

O mecanismo que executa os *bytecodes* é o núcleo da implementação de uma máquina virtual e este pode ser implementado de quatro maneiras diferentes:

Interpretador: é uma emulação em software da máquina Java, na qual um laço de execução busca, decodifica e executa os *bytecodes* até que o programa termine. Por se tratar de uma emulação, esta implementação introduz um esforço adicional e executa mais instruções do que apenas os *bytecodes*.

Compiladores Just-in-time (JIT): é um modelo que tenta acelerar a execução compilando métodos Java em código nativo durante a execução e guardando esta seqüência nativa. Em referências futuras esta seqüência nativa é executada diretamente, sem a necessidade de interpretação. Desvantagens deste modelo são o aumento de duas a três vezes no tamanho do código do objeto, o que é crítico em sistemas embutidos com restrição de memória além de inibir otimizações agressivas pois a compilação só pode gerar um pequeno *overhead*.

Compiladores offline de *bytecodes*: existem duas classes destes compiladores: os que geram código nativo e aqueles que geram uma linguagem intermediária, como C. A geração de código intermediário permite que seja explorada a extensa tecnologia de compilação disponível em diferentes plataformas para a geração de código nativo. A geração de código nativo compromete diretamente a portabilidade e apenas aplicações que não mudam freqüentemente e operam em um sistema homogêneo se beneficiam deste tipo de técnica.

Um processador Java: é um modelo de execução que implementa a JVM diretamente em silício. Este modelo evita todos os *overheads* descritos nos modelos anteriores além de poder ser otimizado para obter uma performance superior em processamento de pilha, *multithreading*, *garbage collection*, endereçamento de objetos e resolução de símbolos. A desvantagem desta técnica é que este tipo de processador ainda não é muito comum em sistemas comerciais multi-uso.

Cada uma dessas implementações possui suas vantagens e desvantagens e são mais ou menos dependentes de uma arquitetura especializada para suportá-las. O objetivo deste

trabalho é otimizar o desempenho da máquina virtual com um mínimo de dependência de arquitetura e impacto na portabilidade desta, por isso nossa escolha foi utilizar uma arquitetura implementada através de um interpretador, portanto nossa implementação de referência será a J2ME CLDC version 1.0.4 Reference Implementation [14]. Mais detalhes sobre esta implementação são apresentados a seguir.

A Implementação de Referência da Sun

O mecanismo de execução de *bytecodes* da KVM é um interpretador. O laço principal deste é dividido em duas funções distintas, o FastInterpret(), ou interpretador rápido, e o SlowInterpret(), ou interpretador lento.

Para continuar a descrição da implementação do interpretador, temos que dividir os *bytecodes* em alguns grupos distintos:

***Standard Bytecodes* (SB):** são os *bytecodes* padrão da máquina virtual pertencentes aos grupos: Load, Store, Stack, ALU, Branches e Jump, descritos na tabela Tabela 2.1.

***Infrequent Standard Bytecodes* (ISB):** também são *bytecodes* padrão da máquina virtual porém possuem uma menor freqüência de ocorrência. Pertencem aos grupos: Constant Pool e Method Call da tabela Tabela 2.1.

***Fast Bytecodes* (FB):** são versões rápidas dos ISB para acesso à métodos e atributos locais à classe em execução. Estes *bytecodes* são gerados pelo próprio interpretador durante a execução, em substituição aos ISB.

O interpretador rápido e o interpretador lento são separados pois o interpretador rápido utiliza cópias locais das principais variáveis do interpretador e quando uma rotina que requer acesso à essas variáveis precisa ser chamada então as cópias locais são escritas de volta aos seus repositórios globais antes de chamar o interpretador lento, onde estão localizados os ISB.

É importante também destacar que durante a execução de um programa, as seguintes operações podem ocasionar uma troca no contexto de execução:

Reescalonamento de *threads*: esta operação pode ocorrer no final de *bytecodes* do tipo Branch, Jump e Constant Pool e Method Call e ocasionam o empilhamento do

contexto de execução atual e a restauração do contexto de execução da *thread* a ser executada.

Chamada de um novo método: cada método possui sua pilha propria de execução, portanto cada vez que um novo método é chamado, o contexto de execução do método atual é empilhado e uma pilha é criada para o novo método.

Apresentadas as arquiteturas da KVM e de seu interpretador, temos agora que comparar suas características com a de uma JVM para saber se podemos aplicar técnicas inicialmente propostas para esta última na KVM. Essa comparação é apresentada nas seções a seguir.

3.2 Caracterização da KVM

Por se tratar de um ambiente projetado para sistemas com restrições de recursos e espaço, precisamos caracterizá-lo antes de comparar ou utilizar qualquer técnica desenvolvida para sistemas desenvolvidos para desktop. Apresentamos então uma caracterização do nosso sistema com base na freqüência dos *bytecodes* e dos tipos de *bytecodes* durante a execução dos *benchmarks*.

O gráfico da Figura 3.2, gerado a partir da execução do conjunto de programas de *benchmark* na implementação de referência, mostra no eixo x o número de *bytecodes* acumulados, em ordem decrescente de freqüência, que correspondem à porcentagem do total de bytecodes executados apresentada no eixo y. Pode ser observado que, assim como em sistemas projetados para desktop (JVM), um pequeno número de *bytecodes* representa a grande maioria dos *bytecodes* executados. Ainda podemos notar que esta distribuição independe do tipo de programa executado.

A Tabela 3.1 apresenta, para cada conjunto de *benchmarks*, o número de *bytecodes* cujas freqüências somadas representam 80% e 90% do número total de *bytecodes* executados. Por ela notamos que uma pequena quantidade destes representa a maioria dos *bytecodes* executados, em média 25 para 90% e 16 para 80% independentemente do programa executado, esta característica também foi observada por Radhakrishnan em [29] para a JVM.

Finalmente, a tabela Tabela 3.2 mostra que as instruções do tipo Load constituem de

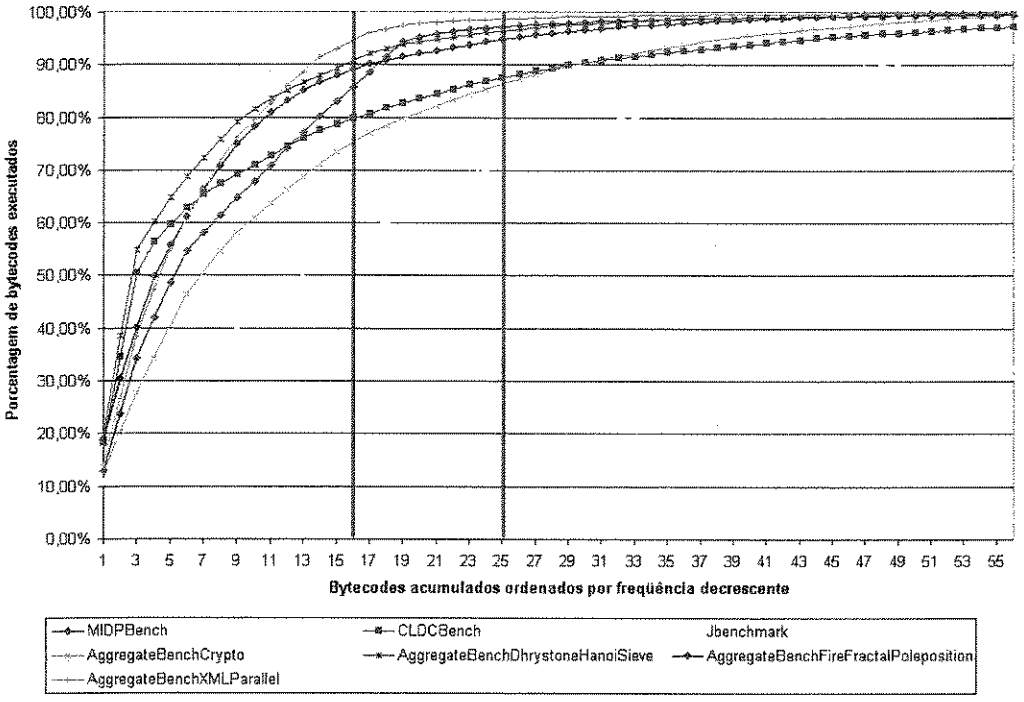


Figura 3.2: Freqüência de ocorrência dos *bytecodes* na execução dos *benchmarks*

34% a 47% das instruções executadas nos *benchmarks*, característica também observada por Radhakrishnan em [29].

Pelos dados apresentados podemos notar que as características da máquina virtual independem da sua plataforma portanto, apenas levando em consideração as restrições de recursos da KVM, podemos aplicar as técnicas de otimização desenvolvidas para JVMs nesta e os resultados devem ser bastante similares.

Estudaremos na próxima seção alguns aspectos de utilização de pilha e freqüência de pares de *bytecodes* na KVM que serviram como base para as propostas de otimização desenvolvidas neste trabalho e apresentadas no Capítulo 4.

Benchmark	90%	80%
MIDPBench	19	14
CLDCBench	30	17
Jbenchmark	48	33
AggregateBenchCrypto	30	20
AggregateBenchDhrystoneHanoiSieve	16	10
AggregateBenchFireFractalPoleposition	18	11
AggregateBenchXMLParallel	14	11

Tabela 3.1: Número de *bytecodes* representando 90% e 80% dos *bytecodes* executados

Benchmark	Loads
MIDPBench	40,4%
CLDCBench	34,1%
Jbenchmark	37,6%
AggregateBenchCrypto	36,0%
AggregateBenchDhrystoneHanoiSieve	41,4%
AggregateBenchFireFractalPoleposition	34,2%
AggregateBenchXMLParallel	47,9%

Tabela 3.2: Proporção dos *bytecodes* do tipo Load em uma execução dinâmica

3.3 Oportunidades de Otimização

Acessos à memória em arquiteturas orientadas a pilha sempre foram uma preocupação e alvos de estudos devido ao seu alto custo no tempo de processamento. Os trabalhos de VanDrunen [32] e Koopman [20], por exemplo, podem confirmar esta afirmação. Na filosofia destes trabalhos foi desenvolvida nossa primeira proposta de otimização cuja idéia principal é a de fazer uma *cache* em registradores dos elementos mais acessados da pilha, uma metodologia similar já foi proposta por Ertl [5], para uma JVM completa. A JVM é baseada na arquitetura de uma máquina de pilha portanto toda passagem de operandos é feita através da memória e, de forma a minimizar esses acessos, estudamos o comportamento dinâmico da passagem de operandos entre *bytecodes*. A Tabela 3.3 mostra o número de pares de *bytecodes* que executam um *push* seguido de um *pop* através da relação entre o número de pares de *bytecodes* executados e a ocorrência de *pushes* seguidos de *pops*, podemos citar como exemplo o conjunto de *benchmarks* AggregateBenchXMLParallel onde, em aproximadamente 70 milhões de *bytecodes* executados, tivemos aproximadamente 40 milhões de pares *push+pop*, ou seja, 57,93% destes pares executam *pushes* seguidos de *pops*. Por esta tabela podemos notar que uma proporção bastante significativa (em média 45%) dos pares de *bytecodes* executados manipula apenas um elemento da pilha. Este comportamento comprova o estudo de Ertl [5] que sugere que a proporção de pares de *bytecodes* que manipulam mais de um elemento da pilha é menos significativa em relação aos que manipulam apenas um.

Benchmark	Número de pushs+pop	Pares de *bytecodes* executados	% de pares executando push+pop
MIDPBench	18.569.832	42.908.090	43,28%
CLDCBench	8.618.957	21.157.258	40,74%
JBenchmark	4.406.766	10.694.506	41,21%
AggregateBenchCrypto	12.346.635	27.449.382	44,98%
AggregateBenchDhrystoneHanoiSieve	3.552.486	8.056.366	44,10%
AggregateBenchFireFractalPoleposition	15.520.656	33.915.531	45,76%
AggregateBenchXMLParallel	40.943.289	70.679.929	57,93%

Tabela 3.3: Caracterização da KVM quanto ao número de *pushes* e *pops* executados

A outra proposta de otimização desenvolvida neste trabalho se baseia nos dados apresentados por O'Donoghue [28] e no conceito de super-instrução apresentado por Prabhas [2]. O'Donoghue [28] faz um levantamento da freqüência de pares de *bytecodes* durante a execução dos *benchmarks* Java Grande Benchmark Suite e SPEC JVM98 Benchmark Suite na Kaffe Virtual Machine, versão 1.0.6. Fizemos o mesmo tipo de levantamento para a KVM nos nossos *benchmarks* (Seção 4.1.2) e os resultados são apresentados na Figura 3.3.

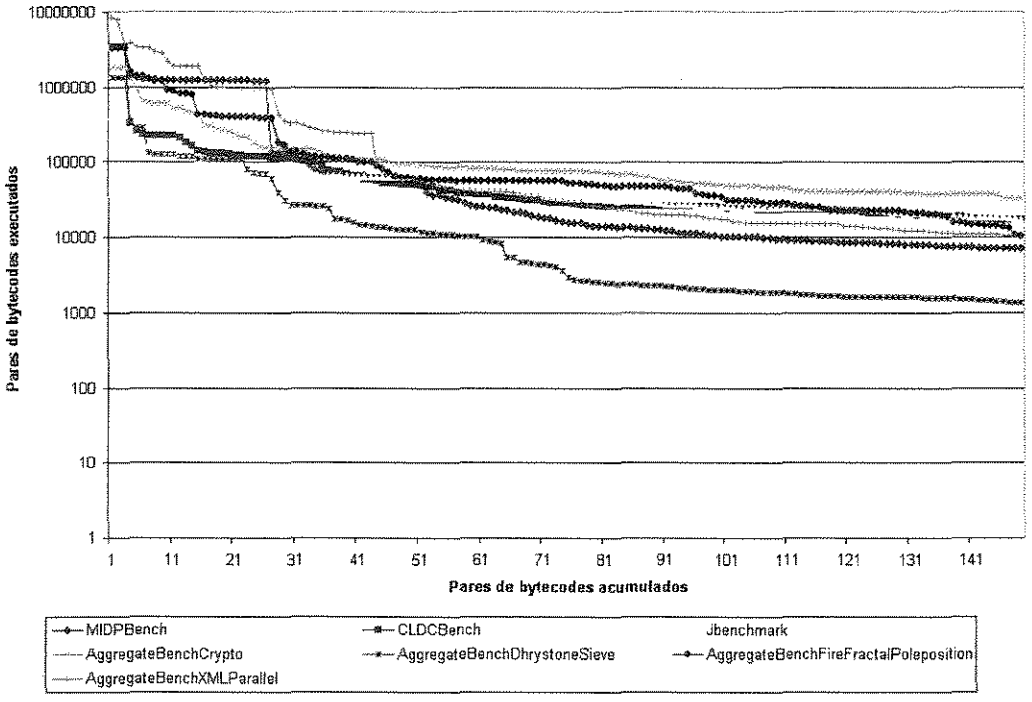


Figura 3.3: Freqüência de ocorrência dos pares de *bytecodes* na execução dos *benchmarks*

O gráfico da Figura 3.3 assim como as observações de O'Donoghue [28] confirma que, independentemente da plataforma ou do *benchmark* analisado, um pequeno número de pares de *bytecodes* representa a maior parte dos pares executados. Estes pares mais freqüentes serão o alvo da análise da segunda proposta de otimização apresentada no Capítulo 4, onde será aplicado o conceito de super-instrução apresentado por Prabhas [2] nestes pares.

Neste capítulo foram apresentadas uma breve descrição da arquitetura da KVM além

de uma caracterização desta, a partir da qual analisamos a KVM sob os aspectos de utilização da pilha e freqüência de pares de *bytecodes* compondo os dados que motivaram a implementação das duas técnicas de otimização deste trabalho. No próximo capítulo serão descritos estes métodos de otimização bem com os resultados obtidos pela aplicação destes na implementação de referência.

Capítulo 4

Técnicas de Otimização

Com base nos dados apresentados no Capítulo 3 e em técnicas de otimizações já aplicadas à JVM apresentadas no Capítulo 2, implementamos duas técnicas de otimização na KVM. O ambiente de desenvolvimento e os parâmetros de comparação utilizados para avaliar essas técnicas de otimização são descritos na Seção 4.1. A primeira técnica é baseada na redução do número de acessos à memória através da redução dos acessos à pilha e é apresentada na Seção 4.2. A outra técnica foi implementada através da fusão de *bytecodes* em uma única instrução baseada na freqüência desses *bytecodes*, detalhes sobre esta implementação podem ser vistos na Seção 4.3.

4.1 Ambiente de Desenvolvimento e Parâmetros de Comparação

4.1.1 Ambiente de Desenvolvimento

Como base para o desenvolvimento das técnicas propostas foi utilizada a KVM da implementação de referência da Sun para a CLDC 1.0.4 [14] . Para execução dos *benchmarks* comerciais utilizamos, como camada de perfil, a implementação de referência da Sun do MIDP 1.0.3 para Solaris [15]. A máquina utilizada para rodar os *benchmarks* é uma UltraSPARC-IIi com 1 processador de 333 MHz e 256 Mb memória RAM, rodando SunOS 5.6.

4.1.2 Parâmetros de Comparação

Utilizamos como parâmetros de comparação o tamanho do código objeto gerado por cada uma das propostas e também a variação em relação à implementação de referência da Sun nos resultados dos seguintes *benchmarks*:

WSAS J2ME Benchmark Suite: é um conjunto de *benchmarks* desenvolvido pela Motorola que avalia os aspectos da linguagem Java específicos para CLDC, métodos das bibliotecas da CLDC, MIDP 1.0a e MIDP 2.0 além de alguns *benchmarks* agregados. *Benchmarks* agregados são aqueles que usam algoritmos padrões de *benchmark* ou são adaptações de MIDlets funcionais. As saídas deste *benchmark suite* são:

CLDCLang: este subconjunto mede a performance das funcionalidades Java requeridas pela especificação da CLDC. Seus *benchmarks* são:

ArrayBench: manipula diversos tipos de *arrays* de objetos.

ByteBench: executa operações matemáticas usando *bytes*.

IntBench: executa operações matemáticas usando inteiros.

LogicalBench: manipula variáveis booleanas.

LongBench: executa operações matemáticas usando *longs*.

LoopBench: executa loops *while()* e *for()* vazios.

MethodBench: invoca uma variedade de diferentes tipos de métodos.

CLDCLib: este subconjunto mede a performance de funções de bibliotecas específicas requeridas pela especificação do CLDC. Seus *benchmarks* são:

LoadBench: carrega dinamicamente uma classe de um arquivo jar e executa o inicializador estático da classe.

IOBench: manipula streams de entrada e saída.

MultiThreadBench: mudanças entre múltiplas threads.

StringBench: executa métodos da classe java.lang.String.

SystemBench: executa métodos da classe java.lang.System além de gerar exceções (throw/catch).

VectorBench: executa métodos da classe java.util.Vector.

MIDPLib: MIDPLib: este subconjunto mede a performance dos métodos gráficos e RMS da biblioteca MIDP. Seus *benchmarks* são:

DialogBench: executa alguns métodos das classes javax.microedition.lcdui.TextField e javax.microedition.lcdui.TextBox.

GraphicsDraw: executa métodos primitivos de desenho para um *buffer* fora da tela da classe javax.microedition.lcdui.Graphics.

GraphicsFont: executa métodos de manipulação de fontes das classes javax.microedition.lcdui.Graphics e javax.microedition.lcdui.Font.

GraphicsImage: executa métodos para criação e desenho de objetos da classe javax.microedition.lcdui.Image para um buffer fora da tela.

GraphicsRepaint: executa métodos de renderização da classe javax.microedition.lcdui.Canvas.

RMSRebuildBench: executa métodos para avaliar a capacidade de reconstrução e *reset* da classe javax.microedition.rms.RecordEnumeration.

RMSStoreBench: executa métodos de adição, manipulação e deleção da classe javax.microedition.rms.RecordEnumeration.

Aggregate: são os *benchmarks* que executam um algoritmo conhecido ou que executam uma aplicação Java que possibilita a medição de tempos.

DhrystoneBench: implementação Java do *benchmark* Dhrystone.

HanoiBench: usa um algoritmo padrão do Navy's Ocean and Surveillance Center para resolver o problema das Torres de Hanoi para um número variável de discos.

SieveBench: usa o algoritmo da Peneira de Eratosthenes para determinar os números primos entre 1 e 512.

XMLBench: usa o analisador kXML para analisar um arquivo XML.

ParallelBench: usa algoritmos multi-tarefas de multiplicação de matrizes e *merge sort*.

CryptoBench: codifica e decodifica textos usando os algoritmos DES, triple DES, IDEA, Blowfish e Twofish.

FireBench: desenha uma imagem na tela.

FractalBench: criação de um fractal Mandelbrot.

PolePositionBench: modelo de um jogo MIDP 1.0a.

Jbenchmark [18]: é um *benchmark* público que mede a performance gráfica de dispositivos J2ME através de 5 testes, com 10 segundos de duração cada. As saídas deste *benchmark* são:

Text: rotinas de manipulação de textos.

2D shapes: rotinas de manipulação de formas 2D.

3D shapes: rotinas de manipulação de formas 3D.

Fill rate: rotinas de preenchimento de áreas

Animation: rotinas de manipulação de animações.

Exceto para o Jbenchmark [18], que é público, nenhuma referência para estes *benchmarks* é apresentada pois o código fonte deles é de propriedade da Motorola Inc. Esses *benchmarks* só são concedidos para licenciados da MERI (Motorola Embedded Reference Implementation) [26] e foram concedidos para esta tese sob a condição de não divulgação do código fonte nem de documentos confidenciais.

Para cálculo do ganho geral de um conjunto de *benchmarks* utilizamos a fórmula 4.1 que define o ganho geral como a porcentagem de aumento na velocidade de execução total do *benchmark*.

$$Ganho\ Geral = \frac{\sum Ganho\ de\ cada\ teste(em\ segundos)}{numero\ de\ testes} \tag{4.1}$$

4.2 Redução dos Acessos à Pilha

A primeira proposta de otimização deste trabalho tem como objetivo a redução do número de acessos à memória através de uma adaptação da implementação da pilha da KVM para que elementos desta sejam mantidos em registradores em determinadas situações. Principalmente em uma arquitetura RISC, em que não existem instruções que manipulam dados diretamente na memória, a redução de operações do tipo *load* e *store* podem representar um ganho significativo no tempo de execução de um programa.

A implementação da pilha na KVM é representada por uma área contínua de memória e por um apontador para o topo desta, como ilustrado na Figura 4.1.

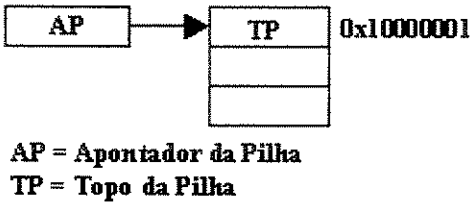


Figura 4.1: Implementação original da pilha para a KVM

A implementação desta técnica consiste em manter um elemento do topo da pilha (TOS) em um registrador eliminando assim, no melhor caso, um acesso de leitura e um acesso de escrita à pilha. A implementação da pilha ficaria como ilustrado na Figura 4.2.

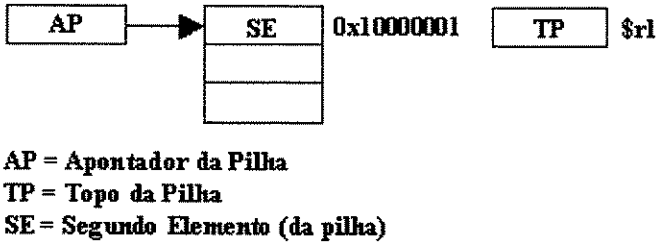


Figura 4.2: Implementação modificada da pilha para a KVM

A escolha de manter apenas um dos elementos da pilha em registrador foi baseada nas seguintes premissas:

- Limitação do aumento do código objeto: por se tratar de um sistema com limitações de recursos temos que ter cuidado com a quantidade de código adicionado. Para cada elemento que mantivéssemos em registrador, um novo estado no laço do interpretador teria que ser adicionado. Como veremos a seguir, um novo estado significa

a duplicação de todos os *bytecodes* do laço principal do interpretador (interpretador rápido).

- A proporção de pares de *bytecodes* que executam um *pop* na seqüência de um *push* é bastante significativa, em média 45% conforme visto na Tabela 3.3 e, de acordo com Ertl [5], a proporção dos que executam mais de um *pop* ou *push* seguidos é muito menor.

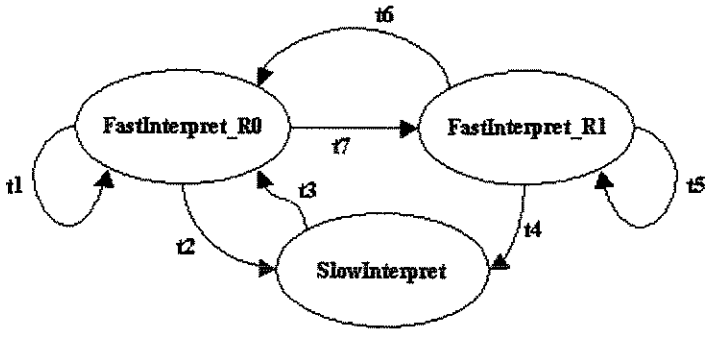


Figura 4.3: Máquina de estados para o cache de 1 elemento

A Figura 4.3 representa a máquina de estados da nossa implementação e seus estados são:

- FastInterpret_R0: é o estado em que não temos nenhum dado no registrador e caso o *bytecode* faça o *pop* de um valor, deve buscá-lo diretamente na pilha. Já no caso em que o *bytecode* fosse fazer o *push* de um valor na pilha, deve agora colocá-lo no registrador.

- FastInterpret_R1: é o estado onde os *bytecodes* são modificados para buscar o primeiro elemento da pilha no registrador e, sempre que possível, escrever o resultado de sua operação nele.

- SlowInterpret: este estado permanece inalterado em relação à implementação original. Antes da transição para este estado a pilha é restaurada ao seu formato original.

Suas transições são descritas a seguir:

- t1: *bytecode* consome um ou mais elementos da pilha ou não a manipula.
- t2: *bytecode* implementado no interpretador lento.
- t3: fim da execução de um *bytecode* lento.
- t4: *bytecode* implementado no interpretador lento.
- t5: *bytecode* adiciona um elemento ao registrador ou não o consome.
- t6: *bytecode* consome o elemento do registrador.
- t7: *bytecode* adiciona um elemento ao registrador.

Note que não há transição do estado SlowInterpret para FastInterpret_R1. Isto ocorre porque os *bytecodes* deste estado possuem vários pontos de saída e, em cada *bytecode*, no mínimo um destes pontos pode gerar um reescalonamento de *threads*, onde é necessário ter a pilha no seu formato padrão.

Na transição do estado FastInterpret_R1 para o estado SlowInterpret (t4) é necessário, além da restauração das cópias locais das variáveis do interpretador, que também ocorre na transição de FastInterpret_R0 para SlowInterpret (t2), uma restauração da pilha para seu formato padrão pois o TOS também é local aos estados FastInterpret_R0 e FastInterpret_R1.

Os estados FastInterpret_R0 e FastInterpret_R1 estão no mesmo contexto de execução e são implementados por dois *"switches"* independentes. O pseudo-código a seguir ilustra melhor esta implementação:

```
register cell l_regStack;
FastInterpret()
{
FastInterpret_R0: /* Start point of FastInterpret_R0 state */
    switch(*ip)
    {
#include "bytecodes_R0.c"
    }
FastInterpret_R1: /* Start point of FastInterpret_R1 state */
    switch(*ip)
    {
#include "bytecodes_R1.c"
    }
}
```

A implementação dos *bytecodes* consiste em, para cada um dos estados, considerar a presença ou não de um valor no registrador e também a existência ou não de valor de retorno do *bytecode*. Tomemos como exemplo a seguir a implementação de alguns dos *bytecodes*:

Caso 1: *bytecode* consome um elemento da pilha.

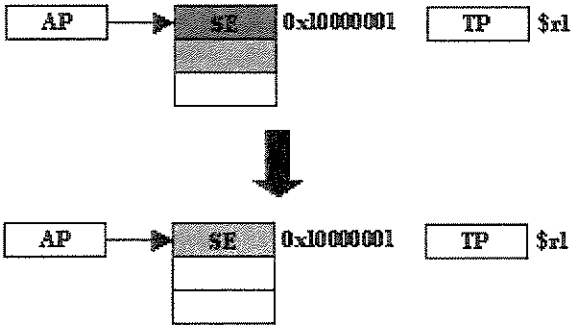


Figura 4.4: Pilha - caso 1, estado FastInterpret_R0

No caso descrito na Figura 4.4, não há modificação do código fonte nem transição

de estados pois não há dados no registrador na entrada (estado é FastInterpret_R0) nem na saída do *bytecode* (*bytecode* apenas consome um elemento da pilha e o atribui à uma variável local).

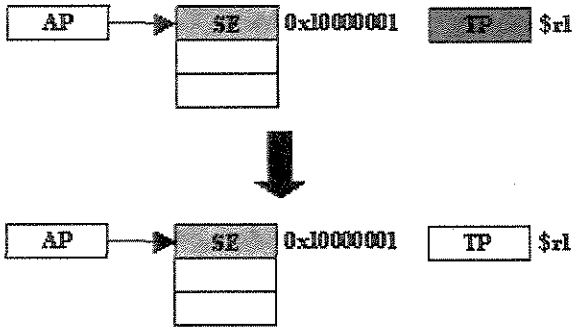


Figura 4.5: Pilha - caso 1, estado FastInterpret_R1

No caso descrito na Figura 4.5, há um elemento no registrador na entrada do *bytecode* (estado é FastInterpret_R1), portanto é economizado um acesso de leitura à pilha buscando o dado diretamente no registrador. A máquina transiciona para o estado FastInterpret_R0 pois o dado que estava no registrador foi consumido.

Caso 2: *bytecode* adiciona um elemento à pilha.

No caso descrito na Figura 4.6, o registrador estava vazio na entrada do *bytecode* (estado FastInterpret_R0), o *bytecode* adiciona um elemento ao registrador e transiciona a máquina para o estado FastInterpret_R1. Economia de uma escrita à memória.

No caso descrito na Figura 4.7, na entrada do *bytecode*, o registrador já possui um valor e é necessário movê-lo para a pilha antes de atualizá-lo o registrador com o novo valor. Esta operação adiciona o *overhead* de uma cópia no registrador à execução original do *bytecode*.

Caso 3: *bytecode* consome um elemento da pilha na sua entrada e adiciona um elemento

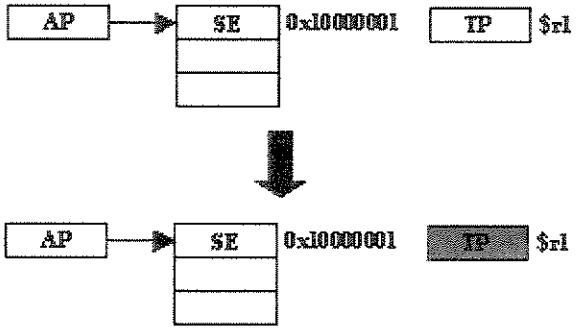


Figura 4.6: Pilha - caso 2, estado FastInterpret_R0

na saída.

No caso descrito na Figura 4.8, o registrador não possui informação na entrada do *bytecode* que coloca seu valor de saída neste. Economia de um acesso de escrita à pilha.

O caso descrito na Figura 4.9 é o melhor para esta técnica de otimização, o *bytecode* consome o dado do registrador na sua entrada e escreve seu valor de saída neste, economizando um acesso de leitura e um de escrita à pilha.

Em termos da implementação dos *bytecodes*, o pior caso é estar no estado FastInterpret_R1 e o *bytecode* seguinte, ao ser executado, empilhar um elemento antes de desempilhar outro (ex: caso 2, estado FastInterpret_R1). Porém, existem outras situações que podem acrescentar um *overhead* na execução do interpretador, são elas:

Reescalonamento de *threads*: *bytecodes* que chamam o *jump* incondicional "goto rescheduleP oint"

podem causar um reescalonamento de *threads* ou uma chamada ao *garbage collector* e, em qualquer um destes casos, a pilha precisa estar em seu estado normal para que possamos simplificar nossa implementação e não mexer em outros componentes além do interpretador. Nesta situação, ao entrar em um *bytecode* deste tipo quando

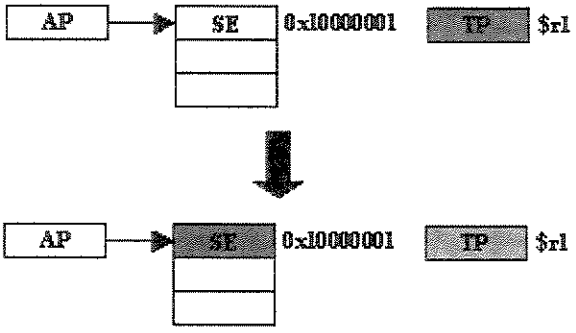


Figura 4.7: Pilha - caso 2, estado FastInterpret_R1

no estado FastInterpret_R1, é necessário restaurar a pilha ao seu formato padrão, introduzindo um acesso adicional à memória na execução deste.

Chamada do interpretador lento: nesta condição todas as variáveis locais ao interpretador rápido devem ser restauradas ao seu repositório local, entre elas o elemento TOS, introduzindo assim um acesso de escrita à pilha.

***Bytecodes* que chamam métodos nativos:** métodos nativos também podem manipular a pilha. Como estes estão fora do contexto do interpretador rápido e também podem acionar o *garbage collector*, precisam ter a pilha restaurada ao seu formato padrão.

Os resultados finais obtidos a para a implementação desta técnica são apresentados na Tabela 4.1 e detalhados no Apêndice A, onde podem são apresentados resultados muito expressivos que chegam a até 37,71% em alguns testes dos programas de *benchmark* analisados. Esses resultados são discutidos a seguir.

Para o conjunto de *benchmarks* MIDPBench podemos notar, pela tabela Tabela 4.2, que há um aumento geral de 17,23% na performance do módulo GraphicsFont. Este módulo, como visto na Seção 4.1.2 lida com a manipulação de fontes e, apesar de não termos acesso ao seu código fonte, podemos deduzir que ele se utiliza principalmente dos

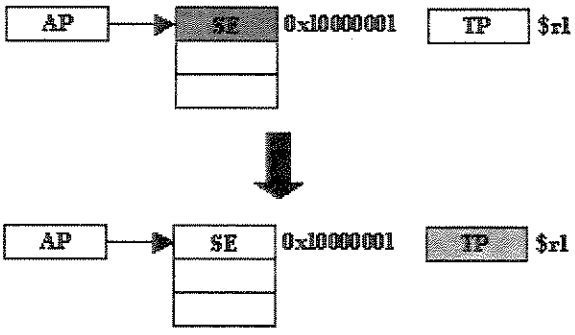


Figura 4.8: Pilha - caso 3, estado FastInterpret_R0

Benchmark Suite Name	Speed up
MIDPBench	3,66%
CLDCBench	5,30%
JBenchmark	1,90%
AggregateBenchCrypto	0,14%
AggregateBenchDhrystoneHanoiSieve	4,51%
AggregateBenchFireFractalPoleposition	5,66%
AggregateBenchXMLParallel	4,81%

Tabela 4.1: Resultados gerais para a técnica de redução de acessos à pilha (RSA)

bytecodes do interpretador rápido.

Alguns módulos dentro deste mesmo conjunto, no entanto, não se beneficiaram da aplicação desta técnica, apresentando inclusive algumas perdas no desempenho. Dentre estes, os mais afetados foram o DialogBench e o GraphicsDraw (Seção A.7). A influência negativa na velocidade de processamento destes *benchmarks* deve-se principalmente à sua natureza, cujo foco é exercitar métodos nativos da biblioteca do MIDP e provavelmente possuam pouco código além de chamadas a esses métodos nativos. Como já descrevemos no início deste capítulo, nossa implementação concentra-se na otimização dos *bytecodes* rápidos portanto chamadas de métodos nativos, localizadas em *bytecodes* lentos, não se

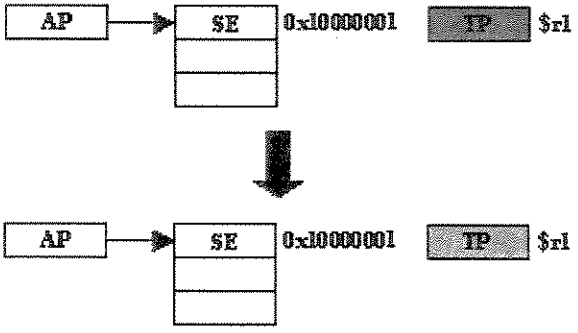


Figura 4.9: Pilha - caso 3, estado FastInterpret_R1

beneficiam dela. Em tempo final de execução, este conjunto de *benchmarks* teve um ganho de 3,66%.

O conjunto de *benchmarks* que apresentou o comportamento mais regular com a aplicação desta técnica foi o CLDCBench (Seção A.6), com um ganho geral de 5,30% em tempo de execução. Este conjunto de *benchmarks* superou a implementação de referência em 61 dos seus 73 testes e entre os resultados mais significativos podemos destacar as operações com tipos *byte* (ByteBench) que obtiveram 9,57% de ganho geral, as operações de *loop* (LoopBench) com ganho geral de 8,65% e as operações com vetores (VectorBench) com ganho geral de 6,94%.

Estes resultados podem ser explicados principalmente pela natureza deste conjunto de *benchmarks* pois boa parte dos testes foram projetados para avaliar a implementação do interpretador, onde os *bytecodes* rápidos são maioria. As perdas de desempenho concentraram-se principalmente nas operações com tipos *long* pois, como cada um destes ocupa duas posições na pilha, não é possível aplicar esta técnica de otimização que só se utiliza de um registrador.

O conjunto de *benchmarks* AggregateBenchDhrystoneHanoiSieve (Seção A.2) também merece destaque pois obteve um ganho geral de aproximadamente 4,51% consistentemente entre seus testes, o que se deve à natureza matemática destes testes que são implementações em Java de alguns algoritmos para solução de problemas matemáticos.

Benchmark Module	Benchmark Name	Speed up
GraphicsFont (1.6)	drawString	7,07%
	drawSubstring	5,90%
	drawChars	13,54%
	drawChar	13,84%
	stringWidth	27,67%
	substringWidth	28,06%
	charsWidth	26,50%
	charWidth	15,28%
Total		17,23%

Tabela 4.2: Resultados para módulo MIDPBench - GraphicsFont

Entre os *benchmarks* mais voltados aos aspectos gráficos da implementação, destacamos os resultados do AggregateBenchFireFractalPoleposition que obteve um ganho geral de 5,66%. O JBenchmark (Seção A.5) obteve apenas um ganho de 1,90%, pois o foco deste é exercitar a resposta gráfica da implementação em análise. A resposta gráfica de um sistema está ligada diretamente à velocidade com que métodos nativos da biblioteca MIDP podem executar, dentre eles estão os métodos do tipo *draw* e métodos da classe Canvas e a maior parte do processamento deste tipo de método está na implementação nativa ao sistema e não na interpretação de *bytecodes*. Esta hipótese pode ser comprovada pelo trabalho de Romer [30] onde dados indicam que alguns programas Java, principalmente os que se utilizam muito de rotinas gráficas, gastam uma grande fração do seu tempo de execução em bibliotecas nativas. O exemplo dado são os *benchmarks hanoi* e *asteroids* onde 57% do tempo de processamento destes ocorre dentro de bibliotecas nativas.

O conjunto de *benchmarks* AggregateBenchXMLParallel (Seção A.4) obteve um ganho de 4,81%, pela descrição de seus testes notamos que são utilizadas múltiplas *threads* para realização de operações como a execução de um algoritmo de *Merge Sort* e multiplicação de matrizes. A troca de contexto de threads ocorre fora do interpretador rápido portanto, antes de cada troca de *thread*, introduzimos o *overhead* de ter que empilhar o elemento TOS. No entanto notamos que, devido à natureza lógica e matemática desses testes (multiplicação de matrizes e ordenação de vetores) que se beneficia da técnica de otimização, o resultado final foi uma superação das perdas ocasionadas pelas mudanças de *thread* pelo ganho no processamento das operações matemáticas.

Em termos de aumento de código objeto, se compararmos com o tamanho do laço original do interpretador rápido, estamos duplicando o tamanho do código fonte porém se compararmos com o tamanho total da plataforma J2ME (aproximadamente 951Kb) ou com a memória ROM total de um telefone celular (entre 8Mb e 16Mb) o aumento do código de 5,9Kb se torna inexpressível, menor que 0,62% e 0,073% respectivamente.

A implementação apresentada nesta seção teve como objetivo provar o conceito de que é possível melhorar a performance da KVM, sem a perda de portabilidade, com um mínimo de aumento do código objeto. Alguns ajustes na implementação poderiam melhorar ainda mais os resultados, dentre eles podem ser destacados:

Suporte às múltiplas pilhas: cada vez que há uma troca de contexto na execução do interpretador é necessário restaurar a pilha ao seu formato original, empilhando o valor armazenado no registrador antes de alocar uma nova pilha. Se, ao invés de apenas um, fossem mantidos um *pool* de registradores o overhead de empilhar o valor do registrador antes da troca de contexto seria reduzido em muitas situações. O valor ótimo para o tamanho deste *pool* dependeria de uma análise da característica dos aplicativos a serem executados nesta máquina baseada na quantidade de chamadas de função empilhadas durante uma execução.

Análise da freqüência dos pares de *bytecodes* executados: quando um *bytecode* empilha um elemento e o próximo, ao invés de consumi-lo, empilha um novo elemento, há o *overhead* de mover o valor do registrador para a pilha antes de atribuir a este o novo valor. Esta situação pode ser amenizada através da análise da freqüência de ocorrência dos pares de *bytecodes* para fazer ajustes nas transições entre os estados de forma a evitar pares de *bytecodes* nesta condição.

4.3 Criação de Super-Instruções

O conceito a ser apresentado nesta seção baseia-se nos trabalhos de O'Donoghue [28] e Prabhas [2]. O'Donoghue [28] demonstrou que assim como a freqüência de ocorrência dos *bytecodes* individualmente, os pares de *bytecodes* também obedecem uma distribuição exponencial, onde poucos pares representam de 80% a 90% dos pares executados. Em [2] Prabhas discute o conceito de super-instrução que baseia-se em extender a máquina vir-

tual com um conjunto de super-instruções, ou combinações de instruções já existentes, escolhidas através de uma análise dinâmica da ocorrência destas seqüências. Um conceito similar é utilizado por Ertl em [7, 8] porém com o enfoque de reduzir erros de predição de *branches* no interpretador.

A diferença entre o método apresentado aqui e o apresentado no trabalho de Prabhas [2] é a de que o dele propõe que as super-instruções sejam criadas através da manipulação dos *bytecodes* do arquivo *class* por uma espécie de pós-compilador, requerendo portanto uma máquina virtual e um pós-compilador dedicados e eliminando a portabilidade do código fonte compilado para esta máquina. Nosso método propõe uma implementação independente de um pós-compilador, capaz de executar o código gerado por um compilador genérico.

Para chegarmos à esta proposta, nos baseamos na seqüência de execução de um *bytecode* que se compõe de: busca, decodificação e execução.

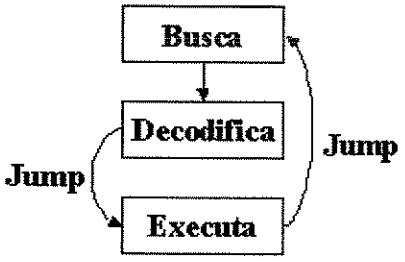


Figura 4.10: Ciclo de execução de um *bytecode*

Pela Figura 4.10 podemos notar que na execução de um *bytecode* temos no mínimo duas operações do tipo *jump* e portanto, na execução de um par de *bytecodes*, teríamos no mínimo quatro dessas operações. Nossa proposta é baseada em, a partir de uma análise dos pares de *bytecodes* mais freqüentes em cada um dos *benchmarks*, tentar eliminar a etapa de decodificação do segundo *bytecode*, compondo então o que chamaremos a partir de agora de **super-instrução**. O ciclo de execução de uma super-instrução ficaria como descrito na Figura 4.11, onde ao invés de fazer um *jump* antes de fazer uma nova busca, esta é feita localmente.

Porém, como nem todo *bytecode* é seguido apenas de um mesmo *bytecode*, no caso da

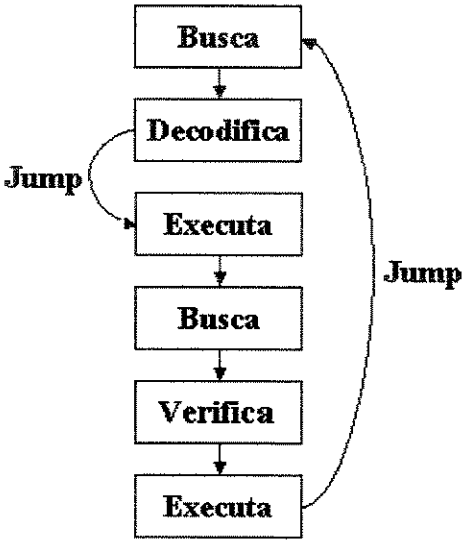


Figura 4.11: Ciclo de execução de uma super instrução - predição correta

decodificação não corresponder ao segundo *bytecode* da super-instrução (predição errada do próximo *bytecode*), adicionaremos o *overhead* de mais uma busca e decodificação ao ciclo, como descrito na Figura 4.12.

Portanto, para a escolha dos pares de *bytecodes* candidatos a serem transformados em super-instruções temos que considerar os seguintes critérios:

Freqüência do primeiro *bytecode*: como estamos reduzindo, no melhor caso, apenas uma instrução do tipo *jump* em relação ao ciclo normal de execução de dois *bytecodes*, para obtermos resultados expressivos e simplificar a implementação temos que escolher *bytecodes* com uma alta freqüência de incidência.

Freqüência relativa do par escolhido: dados empíricos coletados durante os testes mostraram que a freqüência de ocorrência do par escolhido deve ser, no mínimo, maior do que 90% da freqüência do primeiro *bytecode* do par. Esta condição é explicada pelo fato de que, se em menos de 90% das ocorrências do primeiro *bytecode* estaremos acertando seu sucessor e economizando 1 operação do tipo *jump*, na outra

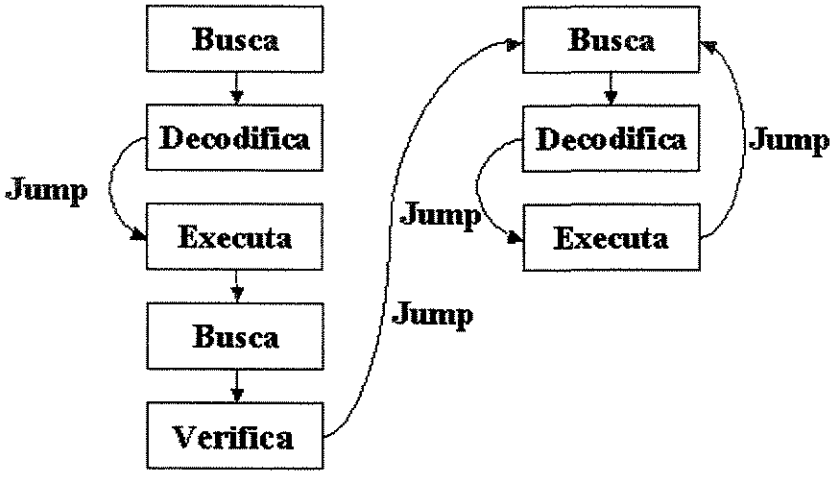


Figura 4.12: Ciclo de execução de uma super instrução - predição errada

parte estaríamos adicionando o overhead de mais um passo de busca e decodificação, o que resultaria em um saldo negativo para a tentativa de otimização.

Tipo do primeiro *bytecode*: *bytecodes* que realizam *jumps* condicionais ou incondicionais não são candidatos a se tornarem super-instruções. No caso de *jumps* condicionais, o *bytecode* subseqüente depende da avaliação da condição e varia de acordo com o estado de execução do programa. No caso de *jumps* incondicionais estes, na sua totalidade, executam alguma operação como reescalonamento, *garbage collection* ou chamada de método nativo antes de executar o *bytecode* seguinte.

A ocorrência dos pares de *bytecodes* na KVM, assim como observado por O'Donoghue [28] obedece uma distribuição exponencial (ver Figura 3.3). Os pares mais freqüentes de *bytecodes*, por conjunto de *benchmarks*, são descritos no Apêndice B.

Aplicamos os critérios de escolha de pares de *bytecodes* em dois *benchmarks* mais genéricos (MIDPBench e CLDCBench) e no AggregateBenchXMLParallel que apresenta dois pares de *bytecodes* com freqüências relativas bem altas. Os pares escolhidos entre os 10 mais freqüentes para cada um deles foram:

MIDPBench: ILOAD_3->IFGT, ISTORE->ILOAD e I2B->BASTORE

Benchmark Suite Name	Speed up
MIDPBench	-0,37%
CLDCBench	-0,90%
AggregateBenchXMLParallel	5,31%

Tabela 4.3: Resultados gerais para a técnica de criação de super-instruções (SI)

CLDCBench: ILOAD_3->IFGT

AggregateBenchXMLParallel: ICONST_1->IADD e ILOAD_3->IFGT

Para uma exemplificação do critério de seleção, tomemos o conjunto de *benchmarks* AggregateBenchXMLParallel cujos dados de freqüência de ocorrência de pares de *bytecodes* e *bytecodes* individualmente são mostrados na Tabela B.4. Os pares de *bytecodes* escolhidos são os 10 mais freqüentes portanto todos satisfazem o primeiro critério de freqüência do primeiro *bytecode*. Aplicando o critério de freqüência relativa do par escolhido, eliminamos os pares ILOAD_1->ICONST_1, ILOAD_1->IALOAD, IINC->ILOAD_3, IADD->IALOAD, IALOAD->ALOAD_0. Aplicando agora o critério de tipo do primeiro *bytecode* eliminamos os pares ALOAD_0->GETFIELDP_FAST, GETFIELDP_FAST->ILOAD_1, IFGT->IINC e IF_ICMPLT->ALOAD_0. Restando então os pares ICONST_1->IADD e ILOAD_3->IFGT como candidatos à super-instruções.

Os resultados obtidos pela implementação desta técnica são apresentados na Tabela 4.3, detalhados nas seções: Seção A.4, Seção A.6 e Seção A.7; e são discutidos a seguir.

O conjunto de *benchmarks* que mais se beneficiou desta técnica foi o AggregateBenchXMLParallel, apresentando um ganho geral de 5,31%, como pode ser visto na Tabela 4.4. Algumas características que podem ser observadas nas duas super-instruções criadas para este módulo é que ambas evitam um acesso à pilha e as freqüências relativas do pares em relação ao primeiro *bytecode* são de 90% e 98%, o que minimiza o número de predições erradas nestas super-instruções.

Para o MIDPBench tivemos um ganho geral negativo de -0,37%, alguns de seus módulos, como o DialogBench e o GraphicsImage, obtiveram ganho. Porém 16 de seus 35 testes apresentaram resultados negativos, o que se deve à baixa freqüência relativa dos

Benchmark Name	SI-2i
Parse XML	0,94%
MergeSort(2 threads)	10,43%
MergeSort(4 threads)	11,64%
MergeSort(8 threads)	10,97%
MergeSort(16 threads)	9,90%
Matrix Multiplication(2 threads)	1,79%
Matrix Multiplication(4 threads)	2,90%
Matrix Multiplication(8 threads)	-0,99%
Matrix Multiplication(16 threads)	0,28%
Total	5,31%

Tabela 4.4: Resultados para o módulo AggregateBenchXMLParallel

pares ILOAD_3->IFGT (71,5%) e ISTORE->ILOAD(88,7%) escolhidos e comprova a validade do critério de escolha baseado em freqüência relativa. Para o CLDCBench tivemos um ganho geral negativo de -0,90%, apesar da freqüência relativa do par escolhido para ele ser alta (96%), a freqüência do primeiro *bytecode* é baixa em relação ao total de pares de *bytecodes* executados portanto este causa pouca influência no resultado geral.

Pelas colunas SI-1i, SI-2i e SI-3i das tabelas das seções Seção A.6, Seção A.4 e Seção A.7 respectivamente, podemos notar um efeito desta técnica de otimização no resultado final, enquanto alguns dos testes são beneficiados pelas super-instruções, outros são prejudicados pelos erros de predição, se tomarmos como exemplo o CLDCBench temos 54 dos seus 73 testes apresentando perdas, no entanto o resultado geral é equilibrado pela freqüência relativa alta do par escolhido, o que sugere uma distribuição não uniforme desse par ao longo dos testes. Outra característica que pode ser observada pelas tabelas do Apêndice B é a de que alguns pares de *bytecodes* são comuns entre os mais executados nos diversos *benchmarks*.

Em termos de aumento de código fonte esta técnica é bastante eficiente, para o CLDC-Bench o aumento no código objeto foi de apenas 50 bytes, para o AggregateBenchXMLParallel 80 bytes e para o MIDPBench 136 bytes, representando 0,0052%, 0,0084% e 0,0142% respectivamente em relação ao tamanho total da plataforma J2ME, que é de 951Kb.

Pelos resultados apresentados podemos concluir que esta técnica é recomendada, da forma como foi implementada, em casos específicos como o do AggregateBenchXMLPa-

rallel em que o número de predições erradas é bem reduzido ou em situações em que a máquina virtual é utilizada para uma função específica e é aceitável prejudicar a performance de alguns *bytecodes* para melhorar a de outros. Esta proposta sugere ainda que esta característica da máquina virtual pode ser usada por compiladores ou pós-compiladores para a criação de super-*bytecodes*, onde pode ser obtido um aumento na performance a custo muito baixo em termos de código fonte e sem nenhum prejuízo ao desempenho dos outros *bytecodes*.

Capítulo 5

Conclusões e Trabalhos Futuros

O objetivo principal deste trabalho foi a implementação de duas técnicas de otimização, projetadas para uma máquina virtual Java (JVM) completa, no interpretador da K Virtual Machine (KVM). As técnicas foram testadas na implementação de referência da Sun Microsystems Inc. para a Connected Limited Device Configuration (CLDC) obtendo ganhos de 0,14% a 5,66% na primeira delas e -0,90% a 5,32% na outra para um conjunto de *benchmarks* disponíveis para este tipo de plataforma, numa máquina UltraSPARC-IIi.

A técnica de otimização através da redução dos acessos à pilha foi a que apresentou um melhor resultado geral entre os benchmarks testados, apesar de seu melhor resultado ser bastante próximo ao da outra técnica, 5,66% contra 5,32%, ela apresentou resultados mais consistentes entre todos os testes, superando a implementação de referência em 102 de 134 testes e em termos de resultados gerais, superando todos os resultados da implementação de referência. Esta é uma alternativa bastante viável mesmo para sistemas com limitação de espaço em memória pois o aumento do tamanho do código foi de apenas 0,62% se comparado com o tamanho do executável original (aproximadamente 951Kb).

A segunda técnica de otimização foi baseada na redução de alguns passos do ciclo de interpretação de um *bytecode* através da criação de super-instruções. Para uma situação específica de freqüência de pares de *bytecodes* esta técnica resultou em um ganho geral de 5,32% porém em algumas situações apresentou perdas de 0,90% e 0,37%, em termos do número total de testes rodados esta técnica superou a implementação de referência em 46 de 116 testes rodados o que mostra que, exceto no benchmark AggregateBenchXMLParallel em que ela superou 8 de 9 testes rodados, esta técnica é viável em condições onde é aceitável melhorar o desempenho de algumas operações em troca das perdas na perfor-

mance de outros. Em termos de aumento do código fonte esta técnica é mais eficiente do que a primeira variando entre 0,0052% e 0,0142% comparado com o executável original (aproximadamente 951Kb).

Como a segunda técnica apresentou ganho apenas em situações muito específicas e se baseia na associação de pares de *bytecodes*, se tomarmos como exemplo o AggregateBenchXMLParallel alguns acessos à pilha já foram eliminados pela associação destes bytecodes. Por essa razão não tentamos uma combinação dos dois métodos neste trabalho, pois seus ganhos não seriam somados.

5.1 Trabalhos Futuros

Apesar de já apresentar resultados superiores à implementação de referência em todos os *benchmarks* em que foi testada, alguns aspectos da técnica de redução dos acessos à pilha ainda podem ser estudados com o intuito de melhorar ainda mais sua performance, são eles:

- Suporte à múltiplas pilhas: cada vez que há uma troca de contexto na execução do interpretador é necessário restaurar a pilha ao seu formato original, empilhando o valor armazenado no registrador antes de alocar uma nova pilha. Se, ao invés de apenas um, fossem mantidos um *pool* de registradores o overhead de empilhar o valor do registrador antes da troca de contexto seria reduzido em muitas situações. O valor ótimo para o tamanho deste *pool* dependeria de uma análise da característica dos aplicativos a serem executados nesta máquina baseada na quantidade de chamadas de função empilhadas durante uma execução.

- Análise da freqüência dos pares de *bytecodes* executados: quando um *bytecode* empilha um elemento e o próximo, ao invés de consumi-lo, empilha um novo elemento, há o *overhead* de mover o valor do registrador para a pilha antes de atribuir a este o novo valor. Esta situação pode ser amenizada através da análise da freqüência de ocorrência dos pares de *bytecodes* para fazer ajustes nas transições entre os estados de forma a evitar pares de *bytecodes* nesta condição.

No caso da técnica de criação de super-instruções, os resultados mostram que esta idéia pode ser aplicada à compiladores ou pós-compiladores com ganhos bem expressivos em

alguns testes e sem prejuízo ao desempenho de outros *bytecodes*. A distribuição exponencial dos pares de *bytecodes* favorece o controle do aumento de código fonte nesta técnica que pode ser aplicada, por exemplo, apenas aos pares de *bytecodes* mais freqüentes.

Apêndice A

Resultados dos *Benchmarks*

As tabelas apresentadas a seguir referem-se aos resultados da aplicação de cada uma das técnicas de otimização sobre a implementação de referência da máquina virtual.

Legenda:

RI (*Reference Implementation*): resultados do *benchmark* para a implementação de referência em operações/segundo (média de 3 execuções).

RSA (*Reduced Stack Access*): porcentagem de variação da velocidade de execução sobre a implementação de referência para a aplicação da técnica de redução de acessos à pilha.

SI (*Super-Instructions*): porcentagem de variação da velocidade de execução sobre a implementação de referência para a aplicação da técnica de criação de super-instruções. 1i, 2i e 3i representam o número de super-instruções criadas em cada implementação.

A.1 AggregateBenchCrypto

Benchmark Module	Benchmark Name	RI	RSA
CryptoBench (1.2)	DES encryption	44.088	0,16%
	Triple DES encryption	15.352	-0,23%
	IDEA encryption	39.332	-2,39%
	Blowfish encryption	51.091	-1,00%
	Twofish encryption	23.693	-0,60%
Total		173.556	0.14%

A.2 AggregateBenchDhrystoneHanoiSieve

Benchmark Module	Benchmark Name	RI	RSA
DhrystoneBench (1.6)	dhrystone	11.659	3,01%
HanoiBench (1.5)	hanoi (moves/s)	279.100	5,07%
SieveBench (1.7)	sieve	74,9	5,43%
Total		290.833	4,51%

A.3 AggregateBenchFireFractalPoleposition

Benchmark Module	Benchmark Name	RI	RSA
FireBench (1.5)	Fire Effect(fps)	22,40	11,87%
FractalBench (1.3)	Fractal Effect(fps)	5,16	4,91%
PolePositionBench (1.3)	Dashboard (fps)	75,55	0,23%
	Road (fps)	3.426,29	5,61%
Total		3.529,4	5,66%

A.4 AggregateBenchXMLParallel

Benchmark Module	Benchmark Name	RI	RSA	SI-2i
XMLBench (1.3)	Parse XML	9,82	2,85%	0,94%
ParallelBench (1.3)	MergeSort(2 threads)	1.727,57	2,56%	10,43%
	MergeSort(4 threads)	3.394,76	2,19%	11,64%
	MergeSort(8 threads)	6.493,71	3,04%	10,97%
	MergeSort(16 threads)	11.677,67	2,92%	9,90%
	Matrix Multiplication(2 threads)	33,95	5,65%	1,79%
	Matrix Multiplication(4 threads)	33,66	8,66%	2,90%
	Matrix Multiplication(8 threads)	33,04	8,64%	-0,99%
	Matrix Multiplication(16 threads)	32,98	6,80%	0,28%
Total		23437,18	4,81%	5,32%

A.5 JBenchmark

Test Name	RI	RSA
Text	4639	1,03%
2D Shapes	4882	0,49%
3D Shapes	2462	8,65%
Fill Rate	998	-0,80%
Animation	802	0,12%
Total	13783	1,90%

A.6 CLDCBench

Benchmark Module	Benchmark Name	RI	RSA	SI-1i
ArrayBench (1.6)	assignByte	3.242.300,73	5,64%	1,39%
	assignInt	3.274.643,81	3,48%	1,27%
	assignLong	3.039.739,72	1,32%	-0,68%
	assignClassByte	2.403.538,30	4,50%	-2,00%
	assignClassInt	2.400.090,65	3,56%	-1,80%
	assignClassLong	2.285.185,38	2,79%	-1,95%
	createByte	579.233,20	37,71%	-1,57%
ByteBench (1.6)	asgnByte	6.714.815,27	15,67%	-3,54%
	asgnThis	4.319.419,40	3,87%	-2,03%
	incPost	2.774.057,57	15,47%	-1,21%
	incPre	2.819.162,89	13,36%	-2,10%
	add	2.795.981,88	10,00%	-0,68%
	sub	2.804.774,29	10,36%	-0,90%
	mul	2.346.458,69	1,93%	1,60%
	div	1.825.180,27	6,00%	0,26%
	mod	1.774.182,76	5,09%	-1,12%
	cast2int	6.658.569,17	13,96%	-1,48%
IntBench (1.6)	asgnInt	6.644.781,05	17,86%	-2,02%

Benchmark Module	Benchmark Name	RI	RSA	SI-1i
	asgnThis	4.291.896,07	4,23%	-0,64%
	incPost	11.826.053,67	0,46%	-5,29%
	incPre	11.765.895,00	1,33%	-3,76%
	add	3.413.491,36	11,78%	-0,76%
	sub	3.442.034,54	4,22%	-0,58%
	mul	2.771.471,92	-1,11%	0,10%
	div	2.071.763,11	3,15%	-3,18%
	mod	1.914.224,08	7,78%	-1,12%
	cast2long	4.453.188,26	7,01%	-4,68%
LogicalBench (1.7)	asgnBool	7.062.502,74	17,02%	-3,85%
	asgnThis	4.482.712,08	4,48%	-2,06%
	andOP	1.840.630,25	13,90%	-1,15%
	orOP	1.971.608,09	4,66%	-1,62%
	compOP	2.407.064,33	8,96%	-0,99%
LongBench (1.6)	asgnLong	5.808.145,29	0,99%	-0,85%
	asgnThis	3.724.426,01	0,90%	-1,34%
	incPost	3.155.984,18	-3,62%	0,00%
	incPre	3.135.776,20	-7,52%	-0,21%
	add	2.921.370,62	-0,71%	-1,16%
	sub	2.941.521,22	-0,60%	-1,88%
	mul	1.315.031,32	-3,29%	-0,40%
	div	961.126,83	-1,65%	-0,37%
	mod	876.935,71	-6,33%	-1,60%
LoopBench (1.6)	for loop	2.529.311,07	8,96%	-0,79%
	reverse for loop	3.220.462,51	8,61%	-0,18%
	while loop	2.549.265,71	8,40%	-2,11%
MethodBench (1.2)	invokePackageVirtual	1.406.257,12	2,54%	4,50%
	invokeFinalVirtual	1.511.925,75	4,57%	-1,36%
	invokePublicVirtual	1.067.132,94	6,06%	-0,70%
	invokePrivate	1.519.852,44	3,65%	-1,26%
	invokeInterface	1.035.661,25	4,38%	5,18%
	invokeStatic	1.703.739,91	3,61%	0,19%
LoadBench (1.5)	class load (bytes/s)	506.666,66	-5,26%	0,00%
MultiThreadBench (1.6)	2 threads	9.749,28	7,42%	3,79%
	4 threads	5.773,36	8,49%	3,56%
	8 threads	3.174,88	9,11%	4,45%
	16 threads	1.660,91	9,32%	3,89%
StringBench (1.8)	createEmpty	103.305,18	1,87%	-4,31%
	createFull	99.646,74	1,08%	0,18%
	compareDiff	1.758.259,16	0,78%	-1,36%
	compareSame	968.327,58	0,42%	0,02%
	indexOf	1.166.135,17	0,38%	0,38%
	substring	146.915,80	3,03%	-3,00%
	replace	25.108,83	4,00%	0,09%
	charAt	1.869.257,97	-1,18%	-0,24%
SystemBench (1.6)	collectGarbage	2.173,92	-2,37%	-4,72%
	currentTime	1.011.779,61	1,17%	-0,51%
	freeMemory	639.513,07	-4,11%	5,81%
	exception	127.308,95	27,47%	-6,86%
IOBench (1.2)	OstreamByte	259.296,41	7,81%	-0,14%
	OstreamBlock (bytes/s)	11.548.957,48	0,03%	0,79%
VectorBench (1.2)	addElement	266.092,31	4,57%	-4,09%
	insertElementAt	168.838,81	1,03%	-4,84%
	Add/Remove-Search	78.327,39	11,08%	-2,29%
	Add/Remove-Index	292.064,51	11,10%	-4,05%
Total		184.858.910,80	5,30%	-0,90%

A.7 MIDPBench

Benchmark Module	Benchmark Name	RI	RSA	SI-3i
DialogBench (1.7)	textbox char	78.227,68	-2,07%	2,06%
	textbox string	67.074,66	-5,56%	2,14%
	textfield char	67.383,82	-3,75%	2,50%
	textfield string	57.771,56	-2,45%	2,01%
GraphicsDraw (1.7)	Draw Line (Horz)	134.326,81	-2,04%	5,13%
	Draw Line (Vert)	131.374,28	-0,74%	-0,61%
	Draw Line (Diag)	140.614,83	-0,23%	0,14%
	Draw Rectangle	190.698,31	1,60%	-1,91%
	Fill Round Rectangle	95.294,00	-0,52%	-2,31%
	Draw Arc	61.319,08	-0,59%	-3,04%
	Fill Rectangle	165.439,84	-5,44%	-2,89%
	Fill Arc	68.253,80	-2,18%	-0,76%
GraphicsFont (1.6)	drawString	75.852,36	7,07%	-5,33%
	drawSubstring	72.839,36	5,90%	-4,62%
	drawChars	68.456,86	13,54%	-1,41%
	drawChar	85.215,44	13,84%	-4,72%
	stringWidth	298.905,62	27,67%	-13,63%
	substringWidth	282.465,96	28,06%	-13,06%
	charsWidth	285.970,03	26,50%	-11,82%
	charWidth	602.208,01	15,28%	6,36%
GraphicsImage (1.6)	Create Image (PNG byte array)	214,62	-2,68%	0,45%
	Create Image (blank)	5.955,02	8,34%	18,19%
	Create Image (copy)	236,86	0,52%	0,81%
	Full Render	433,06	0,32%	1,01%
	Clipped Render (10%)	434,54	-0,65%	0,11%
	loadImage (bytes/s)	231.537,25	0,45%	1,52%
GraphicsRepaint (1.6)	request repaint	170.649,23	3,64%	1,69%
	repaint cycle (frame rate)	131.599,12	4,04%	3,69%
RMSRebuildBench (1.5)	reBuild	221,64	-0,81%	-0,58%
RMSStoreBench (1.7)	addRecord	1.691,63	-2,36%	4,24%
	getRecord-ID	23.685,65	0,74%	0,48%
	getRecord-buf	25.723,63	0,55%	1,20%
	ListRecordStores	6.397,86	2,28%	2,61%
	deleteRecord	6.496,18	0,17%	-1,52%
	setRecord	5.677,39	-0,44%	-1,08%
Total		3.640.646,09	3,66%	-0,37%

Apêndice B

Freqüências dos Pares de Bytecodes

Par	Freqüência	Bytecode	Freqüência
IINC->ILOAD_3	1.802.268	ILOAD	3.396.288
ILOAD_3->IFGT	1.802.172	ILOAD_3	2.090.091
IFGT->IINC	1.802.037	IINC	2.062.756
ISTORE->ILOAD	1.316.749	IFGT	1.826.372
ILOAD->BIPUSH	910.115	ISTORE	1.687.590
IOR->ISTORE	641.468	BIPUSH	1.668.149
BIPUSH->IUSHR	629.462	IALOAD	1.157.305
GETSTATICP_FAST->ILOAD	609.996	IAND	1.106.262
BIPUSH->IAND	604.033	IUSHR	742.902
IAND->IALOAD	603.648	IOR	681.356
ILOAD->GETSTATICP_FAST	530.924	GETSTATICP_FAST	615.083

Tabela B.1: AggregateBenchCrypto - Freqüência individual e dos pares de bytecodes

Par	Freqüência	Bytecode	Freqüência
IINC->ILOAD_3	1.320.957	IINC	1.578.640
ILOAD_3->IFGT	1.310.647	ILOAD_3	1.515.749
IFGT->IINC	1.310.519	IFGT	1.311.132
ALOAD_0->GETFIELDP_FAST	325.164	ILOAD_2	438.810
GETFIELDP_FAST->ILOAD_2	284.250	ALOAD_0	365.575
ILOAD_2->IALOAD	284.010	GETFIELDP_FAST	326.846
ILOAD_2->ILOAD_1	130.184	IALOAD	287.115
IF_ICMPLT->IINC	126.649	ILOAD_1	271.346
ILOAD_1->IF_ICMPLT	126.602	IF_ICMPLT	155.396
ILOAD_0->ILOAD_1	126.103	ILOAD_0	128.948
IINC->ILOAD_0	126.000		

Tabela B.2: AggregateBenchDhrystoneHanoiSieve - Freqüência individual e dos pares de bytecodes

Par	Freqüência	Bytecode	Freqüência
IINC->ILOAD_3	3.276.970	ILOAD	6.391.739
ILOAD_3->IFGT	3.276.763	IINC	3.834.637
IFGT->IINC	3.276.598	ILOAD_3	3.360.509
ILOAD->SIPUSH	1.544.688	IFGT	3.278.660
BIPUSH->ISHR	1.368.035	BIPUSH	2.034.365
ILOAD->ILOAD	1.365.082	SIPUSH	1.779.421
ISTORE->ILOAD	1.276.542	IMUL	1.761.540
IMUL->BIPUSH	1.196.450	ISTORE	1.519.283
ILOAD->IMUL	1.195.752	ISHR	1.418.901
ILOAD->IADD	877.081	IADD	1.140.656
IADD->ISTORE	855.781		

Tabela B.3: AggregateBenchFireFractalPoleposition - Freqüência individual e dos pares de bytecodes

Par	Freqüência	Bytecode	Freqüência
ALOAD_0->GETFIELDP_FAST	8.327.539	ILOAD_1	9.825.749
GETFIELDP_FAST->ILOAD_1	7.815.155	ALOAD_0	8.929.374
ICONST_1->IADD	3.821.678	GETFIELDP_FAST	8.329.321
ILOAD_1->ICONST_1	3.775.032	IALOAD	6.277.825
IINC->ILOAD_3	3.276.865	IINC	5.542.638
ILOAD_3->IFGT	3.276.744	IADD	4.127.612
IFGT->IINC	3.276.597	ICONST_1	3.902.430
ILOAD_1->IALOAD	2.830.060	ILOAD_3	3.629.347
IADD->IALOAD	2.818.672	IFGT	3.277.764
IALOAD->ALOAD_0	2.137.061	IF_ICMPLT	2.236.261
IF_ICMPLT->ALOAD_0	1.883.787		

Tabela B.4: AggregateBenchXMLParallel - Freqüência individual e dos pares de bytecodes

Par	Freqüência	Bytecode	Freqüência
ILOAD->ILOAD	420.787	ILOAD	1.158.373
ALOAD_0->GETFIELDP_FAST	212.113	ALOAD_0	910.827
ALOAD_0->GETFIELD_FAST	203.885	IADD	349.891
IADD->ISTORE	193.059	INVOKEVIRTUAL_FAST	316.363
IALOAD->IADD	185.027	ALOAD	306.586
ISTORE->ILOAD	170.333	ISTORE	294.800
INVOKEVIRTUAL_FAST->ALOAD_0	168.483	IMUL	290.840
IMUL->ILOAD	155.264	ICONST_0	271.415
ALOAD->ICONST_0	143.001	IDIV	270.172
ICONST_0->IALOAD	136.340	IALOAD	268.917
ICONST_2->IDIV	126.059	GETFIELDP_FAST	234.102
		GETFIELD_FAST	209.763
		ICONST_2	186.441

Tabela B.5: JBenchmark - Freqüência individual e dos pares de bytecodes

Par	Freqüência	Bytecode	Freqüência
IFGT->IINC	3.327.738	IINC	3.836.543
IINC->ILOAD_3	3.312.411	ILOAD_3	3.427.815
ILOAD_3->IFGT	3.304.763	IFGT	3.401.482
ALOAD_0->GETFIELDP_FAST	333.216	ILOAD	1.228.696
ILOAD->ILOAD	253.784	ALOAD_0	697.991
ILOAD->INVOKESPECIAL_FAST	227.000	BIPUSH	576.063
IF_ICMPNE->ILOAD	225.816	ISTORE	403.182
INVOKESPECIAL_FAST->ISTORE	222.861	INVOKESPECIAL_FAST	385.899
ILOAD->IADD	222.561	IADD	357.168
ILOAD->BIPUSH	222.146	GETFIELDP_FAST	339.984
BIPUSH->IF_ICMPNE	221.144	IF_ICMPNE	246.509

Tabela B.6: CLDCBench - Freqüência individual e dos pares de bytecodes

Par	Freqüência	Bytecode	Freqüência
IINC->ILOAD_3	3.291.745	ILOAD	5.496.810
ILOAD_3->IFGT	3.283.134	IINC	4.633.799
IFGT->IINC	3.276.598	ILOAD_3	4.591.178
ALOAD_0->GETFIELDP_FAST	1.399.444	IFGT	3.310.127
ILOAD->IADD	1.325.107	ALOAD_0	2.783.438
IINC->ILOAD	1.280.860	ALOAD_1	2.612.345
ISTORE->ILOAD	1.276.939	ILOAD_2	1.439.478
ILOAD_2->ILOAD	1.265.325	ISTORE	1.438.024
ALOAD_1->ILOAD_2	1.232.809	GETFIELDP_FAST	1.403.485
INVOKEVIRTUAL_FAST->ALOAD_0	1.228.655	IADD	1.378.779
I2B->BASTORE	1.222.397	INVOKEVIRTUAL_FAST	1.346.174
		BASTORE	1.223.101
		I2B	1.222.397

Tabela B.7: MIDPBench - Freqüência individual e dos pares de bytecodes

Referências Bibliográficas

[1] Hsin-Ta Chiao, Shen-Tzay Huang, Scott Hsu-Jing Kao, and Shyan-Ming Yuan. A portable real-time extension set for java. *IEEE Canadian Conference on Electrical and Computer Engineering*, pages 47–52, 2000.

[2] Prabhas Chongstitvatana. Post processing optimization of byte-code instructions by extension of its virtual machine. *Proceedings of the 20th Conference of Electrical Engineering, Bangkok*, 1997.

[3] K. Driesen and U. Hölzle. Accurate indirect branch prediction. *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA98)*, pages 167–178, 1998.

[4] Luke Dykstra, Witawas Srisa-an, and J. Morris Chang. An analysys of the garbage collection performance in sun's hotspot java virtual machine. *Proceedings of 21st IEEE International Performance, Computing and Communications Conference (IPCCC)*, pages 335–339, 2002.

[5] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technische Universitat Wien, 1996.

[6] Martin Anton Ertl and David Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. *Euro-Par 2001*, pages 403–412, 2001.

[7] Martin Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[8] Martin Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen – a generator of efficient virtual machine interpreters. *Software–Practice and Experience*, 32(3):265–294, 2002.

[9] David Gregg and Martin Anton Ertl. Implementing an efficient java interpreter. *High-Performance Computing and Networking (HPCN Europe 2001)*, pages 613–620, 2001.

[10] David Gregg, Martin Anton Ertl, and Andreas Krall. A fast java interpreter. *Java Optimization Strategies for Embedded Systems (JOSES) Workshop at ETAPS'01*, 2001.

[11] Jane Horgan, James Power, and John Waldron. Measurement and analysis of runtime profiling data for java programs. *IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.

[12] Chang-Hsueh A. Hsieh, Marie T. Conte, and Teresa L. Johnson. A java virtual machine design based on hybrid concurrent compilation model. *Proceedings of IEEE CompCon'97*, 1997.

[13] Sun Microsystems Inc. Java 2 platform micro edition (j2me) technology for creating mobile devices, 2000. http://java.sun.com/products/cldc/wp/.

[14] Sun Microsystems Inc. Java 2 platform, micro edition connected limited device configuration (cldc) version 1.0.4 reference implementation, 2001. http://java.sun.com/j2me/download.html.

[15] Sun Microsystems Inc. Java 2 platform, micro edition mobile information device profile (midp) version 1.0.3 reference implementation, 2001. http://java.sun.com/j2me/download.html.

[16] Sun Microsystems Inc. The java hotspot virtual machine, 2001. http://java.sun.com.

[17] Sun Microsystems Inc. Datasheet java 2 platform, micro edition, 2002. http://java.sun.com/j2me/docs/.

[18] Jbenchmark. http://www.jbenchmark.com.

[19] Austin Kim and Morris Chang. Java virtual machine performance analysis with java instruction level parallelism and advanced folding scheme. *Proceedings of 21st IEEE International Performance, Computing and Communications Conference (IPCCC)*, pages 9–15, 2002.

[20] Philip Koopman. A preliminary exploration of optimized stack code generation. *Proceedings of the 1992 Rochester Forth Conference*, 1992.

[21] Andreas Krall. Efficient javavm just-in-time compilation. *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 205, 1998.

[22] Young-Min Lee, Byung-Chul Tak, Hye-Seon Maeng, and Shin-Dug Kim. Real time java virtual machine for information appliances. *IEEE Transaction on Consumer Electronics*, pages 949–957, 2000.

[23] Martin Maierhofer and Martin Anton Ertl. Optimizing stack code. *Forth-Tagung*, 1997.

[24] Martin Maierhofer and Martin Anton Ertl. Local stack allocation. *Kai Koskimies, editor, Compiler Construction (CC'98)*, pages 189–203, 1998.

[25] Seamus McAteer. Java will be the dominant handset platform. Micro Java, 2002. http://www.microjava.com/articles/perspective/zelos.

[26] Motorola. Motorola embedded reference implementation. http://e-www.motorola.com.

[27] Tia Newhall and Barton P. Miller. Performance measurement of dynamically compiled java executions. *Proceedings of the ACM 1999 conference on Java Grande*, pages 42–50, 1999.

[28] Diarmuid O'Donoghue, Aine Leddy, James Power, and John Waldron. Bigram analysis of java bytecode sequences. *Proceedings of the Second Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, pages 187–192, 2002.

[29] Ramesh Radhakrishnan, N Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.

[30] Theodore Romer, Dennis Lee, Geoffrey Voelker, Alec Wolman, Wayne Wong, Jean-Loup Baer, Brian Bershad, and Henry Levy. The structure and performance of interpreters. *Proceedings of the Seventh Int'l Conference of Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 150–159, 1996.

[31] Fridtjof Siebert. Hard real-time garbage collection in the jamaica virtual machine. *Sixth International Conference on Real-Time Computing Systems and Applications*, pages 96–102, 1999.

[32] Thomas VanDrunen, Antony Hosking, and Jens Palsberg. Reducing loads and stores in stack architectures, 2000. Unpublished Manuscript.

[33] Zheng Weimin, Zheng Fengzhou, Yang Bo, and Wang Yanling. A java virtual machine design based on hybrid concurrent compilation model. *36th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Asia'00)*, pages 18–23, 2000.

[34] Qian Yang, Witawas Srisa-an, Therapon Skotiniotis, and J. Morris Chang. Java virtual machine timing probes: A study of object life span and garbage collection. *Proceedings of 21st IEEE International Performance, Computing and Communications Conference (IPCCC)*, pages 73–80, 2002.