Experimentos com Codificação e Compactação no Gerenciamento de Memória da Máquina Virtual Java

Sandro Rigo

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Experimentos com Codificação e Compactação no Gerenciamento de Memória da Máquina Virtual Java

Sandro Rigo[1]

Junho de 2000

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Arnaldo Vieira Moura
  Instituto de Computação - UNICAMP
- Prof. Dr. Rafael Dueire Lins
  Centro de Tecnologia e Geociências - UFPE
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação - UNICAMP

[1] **Apoio financeiro da Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP (proc. 98/06313-9)**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Rigo, Sandro

Sa57e Experimentos com codificação e compactação no gerenciamento de memória da máquina virtual java / Sandro Rigo – Campinas, [S.P. :s.n.], 2000.

Orientador : Guido Costa Souza de Araújo

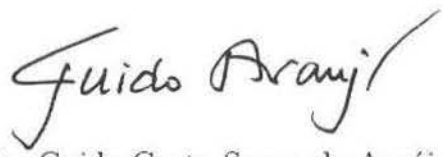
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Linguagem de programação (Computação). 2. Java (Linguagem de programação de computador). 3. Memória de computador. I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Experimentos com Codificação e Compactação no Gerenciamento de Memória da Máquina Virtual Java

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Sandro Rigo e aprovada pela Banca Examinadora.

Campinas, 04 de Julho de 2000.

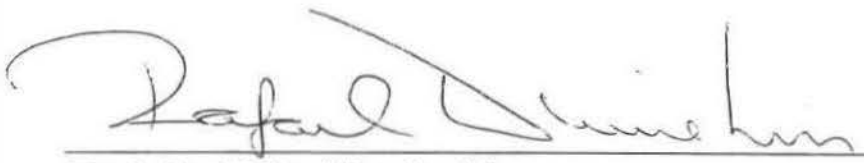


Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

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

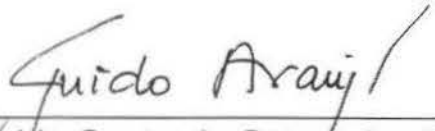
TERMO DE APROVAÇÃO

Tese defendida e aprovada em 04 de julho de 2000, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Rafael Dueire Lins
UFPe

Prof. Dr. Arnaldo Vieira Moura
IC-UNICAMP



Prof. Dr. Guido Costa de Souza Araújo
IC-UNICAMP

Resumo

Gerenciamento de memória é uma tarefa muito importante para o bom desempenho de uma aplicação. Existem duas alternativas para a execução desta tarefa: gerenciamento explícito por parte do programador, ou a existência de um sistema dinâmico automático.

O objetivo desse trabalho foi realizar experimentos de codificação e compactação do *heap* no sistema automático de gerenciamento de memória da máquina virtual Java, usando como plataforma a implementação desta conhecida como *Kaffe*.

Na primeira parte deste trabalho, foi estudado um mecanismo alternativo de codificação dos objetos vivos, baseado em *Binary Decision Diagrams(BDD)*. Na segunda etapa, foi projetado um coletor de lixo, cujo objetivo era avaliar o impacto da compactação do *heap* no desempenho da JVM. A implementação de nosso coletor se baseou no algoritmo conhecido como *Mark-Compact*. Obtivemos uma implementação híbrida para o sistema de recuperação de memória da máquina virtual *Kaffe*, capaz de realizar coletas utilizando ou não compactação. Esse sistema nos permite configurar a frequência com que compactações são realizadas, permitindo ajustá-lo para diminuir o impacto causado por estas no desempenho do programa. Finalmente, mostramos nossas conclusões sobre os experimentos realizados e apontamos algumas possíveis melhorias a serem implementadas na máquina virtual *Kaffe*, que visam aproveitar ao máximo os benefícios trazidos pela compactação.

Abstract

Memory management is an important task for good performance. There are two choices to accomplish this task: explicit management done by the programmer, or the presence of an automatic dynamic memory management system.

The goal of this work was to perform experiments using codification and compaction on Java Virtual Machine's memory management system, using its implementation known as *Kaffe*.

In the first part, we have studied an alternative system of live objects codification, based on *Binary Decision Diagrams(BDD)*. In the second part, a garbage collector has been projected. Its main goal was to evaluate the impact on JVM's performance by adding a heap compaction system. The implementation of this collector was based on the *Mark-Compact* algorithm. We got a hybrid implementation for our garbage collection system, which is able to collect using or not using compaction. This system allows us to configure how often it performs heap compaction, adjusting it to minimize the impact on program performance. Finally, we show our conclusions about the experiments we have done and suggest some possible improvements to be implemented on *Kaffe* JVM, intending to take full advantage of the compaction benefits.

À minha família.

Um homem é um sucesso se pula da cama de manhã,
vai dormir à noite e, nesse meio tempo,
faz o que gosta.

Bob Dylan

Agradecimentos

Mais do que agradeço, mas sim, dedico este trabalho a meus pais, Yara e Wilson, e à minha irmã, Carla. A presença, carinho e apoio que recebo de vocês foram, são e sempre serão essenciais a todas as minhas conquistas.

Agradeço ao professor Guido, pela orientação, amizade e experiência valiosas que me foram passadas durante esses dois anos de convivência.

À FAPESP, pelo apoio financeiro concedido durante o meu mestrado e graduação.

Aos amigos Adreane, Cínthia, Doretto e Vitor pela grande amizade e por tornar a estadia em Campinas mais fácil e divertida.

A todos os amigos do Instituto de Computação, em especial, Bruno, Marcelo, Leonardo, Paulo, Ivan, Nathan, *Maringuetes*, Rodolfo, Márcio, Adilson e Alessandro pela companhia no trabalho e amizade que construímos nesse período.

Ao colega Alexandre Oliva, pelas proveitosas e esclarecedoras discussões ao longo da implementação desse projeto.

À Janaína, Márcio Miranda e Adriano, companheiros de graduação e, acima de tudo, grandes amigos.

À *Transvirtual Technologies* pela criação da máquina virtual *Kaffe* e por manter seu código fonte em domínio público, possibilitando assim, a realização deste e de muitos outros trabalhos científicos.

A Deus, pela saúde, força e inesgotável desejo de aprender.

A todos vocês, deixo um grande abraço e o mais sincero muito obrigado.

Sandro Rigo

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Durante a execução de um programa, objetos são dinamicamente alocados na memória numa região denominada espaço-livre ou *heap*. Embora a quantidade de memória disponível nos computadores venha crescendo consideravelmente ao longo dos anos, este continua sendo um recurso finito e, portanto, é essencial que seja bem administrado.

Podemos entender por gerenciamento de memória o processo de alocação de *recursos de memória* (ou simplesmente *recursos*) disponíveis a cada novo objeto necessário à execução de um programa, e a recuperação dos recursos que estão sendo ocupados por objetos que já não são mais úteis à aplicação em um determinado ponto de sua execução. Esse processo pode ser deixado a cargo do programador, que ficará responsável tanto pela alocação de novos recursos como pela liberação da memória ocupada por objetos que não sejam mais necessários. Para tanto ele deverá utilizar, ao longo do código, chamadas explícitas a procedimentos como *malloc()* e *free()* que estão presentes em várias linguagens (ex. C).

O uso de gerenciamento explícito de memória aumenta a probabilidade de ocorrência de erros que normalmente são muito difíceis de diagnosticar, pois se manifestam somente em condições muito específicas, causando problemas inesperados na execução do programa. Essas condições geralmente são difíceis de reproduzir para depuração e podem não acontecer na fase de testes, e sim após a versão final do programa permanecer rodando normalmente por algum tempo.

Uma das causas deste tipo de problema é a falha na recuperação da memória utilizada por um determinado tipo de estrutura do programa, fazendo com que não haja memória suficiente para o término da execução, caso ela se estenda por um longo período. Outra provável razão para esse tipo de erro é a liberação de um objeto quando ainda existe um apontador para ele. Nesse caso, o comportamento do programa se torna imprevisível, podendo inclusive não apresentar nenhuma falha se o alocador não reutilizar a mesma área de memória para um novo objeto, ou produzir resultados errados se os mesmos dados forem utilizados como sendo de dois objetos diferentes.

A alternativa ao gerenciamento explícito é a existência, no sistema de execução do compilador, de mecanismos que possam identificar os objetos que estão dinamicamente mortos e assim liberar a memória ocupada pelos mesmos automaticamente. Esse mecanismo é conhecido como *Garbage Collector (GC)*.

O *heap*, sob o ponto de vista teórico, é um grafo dirigido G. Os vértices de G são os objetos alocados pelo programa e uma aresta $(x.f_i, y)$ indica que existe um apontador no campo f_i do objeto x, para o objeto y. As raízes[1] de G estão associadas aos objetos apontados pelas variáveis locais do programa em execução. Sendo assim, podemos dizer que um objeto x_n está morto se não existir um caminho $r \rightarrow x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_n$ em G, para qualquer raiz r.

O *Garbage Collector* pode identificar os objetos mortos de duas formas: através de uma contagem direta das referências que cada objeto possui, nesse caso um número de referências igual a zero indica que o objeto está morto, ou atravessando o *heap* e considerando morto um objeto que não seja alcançável por qualquer caminho que se inicia em uma raíz. Algoritmos que adotam essas duas abordagens serão descritos com mais detalhes na Seção 2.1.

A principal virtude de um sistema automático para recuperação de memória é exatamente livrar o programador da obrigação de saber exatamente quais objetos podem ser liberados em um determinado ponto da execução, o que sem dúvida é uma tarefa árdua em sistemas complexos.

A utilização de um sistema de coleta de lixo pode causar um impacto no desempenho da aplicação. Esse impacto vem diminuindo com a elaboração de técnicas mais apuradas de coleta[31] e hoje em dia (*circa* 2000), no caso geral, deve girar em torno de 10% de aumento no tempo de execução em relação ao gerenciamento explícito realizado pelo programador. Esse custo pode ser inteiramente aceitável se considerarmos os benefícios em tempo de desenvolvimento e confiabilidade. Além disso, devido a um possível aumento de localidade de referência gerado pela coleta de lixo, podemos ter casos apresentando uma melhoria no desempenho devido a diminuição de *cache misses* e *page faults*.

1.1 Descrição do Projeto

Com o surgimento da linguagem de programação Java[6], os estudos sobre o problema de *Garbage Collection* receberam um novo impulso. Isso porque essa linguagem requer a existência de um sistema automático de gerenciamento dinâmico de memória em sua máquina virtual [24]. Por esse motivo, escolhemos como plataforma de desenvolvimento para esse projeto a linguagem Java. O principal objetivo deste estudo é avaliar o efei-

[1] Raízes são vértices de onde somente saem arestas

to da incorporação de um sistema que realize codificação e compactação de memória ao coletor de lixo presente na máquina virtual. Para isso vamos utilizar uma implementação da máquina virtual Java(JVM)[2] conhecida como *Kaffe*[3], que aparentemente é a implementação de código de domínio público mais bem sucedida até o presente momento.

No decorrer do projeto foi realizado um detalhado estudo das principais técnicas utilizadas na implementação de sistemas de coleta automática de lixo e da implementação desse sistema na máquina virtual *Kaffe*.

Ao longo da primeira fase do projeto, estudamos a possibilidade de incluir uma codificação dos objetos vivos no *heap*. Esse esquema utiliza o conceito de *Binary Decision Diagrams (BDD)*[3] [12] para codificar os endereços de objetos vivos. BDD é uma representação estrutural de funções booleanas, que tem sido usada com sucesso, na codificaçao de sistemas complexos[13]. A partir da representação em BDD de endereços no heap, extrairíamos informações que facilitariam o processo de compactação dos objetos vivos. Essa codificação acabou não sendo implantada na versão final de nosso coletor, mas o estudo, as implementações realizadas e a avaliação deste estão detalhadamente descritas no Capítulo 3.

A segunda fase do trabalho consistiu na introdução de novos sistemas de alocação e de recuperação de memória no sistema de gerenciamento de memória da JVM *Kaffe*. O objetivo, neste caso, foi tratar o problema de fragmentação de memória e analisar o desempenho desse sistema após a introdução de um algoritmo de compactação dos objetos no *heap*. A implementação mantém a interface de comunicação entre o alocador/coletor e a máquina virtual originalmente presente na *Kaffe*, de modo que o uso de nosso sistema de memória implica apenas na troca de poucos arquivos da distribuição da JVM *Kaffe*.

Vale salientar que, no início da fase de implementação do sistema de compactação fomos obrigados a *congelar* o código fonte da máquina virtual porque a *Kaffe* é um sistema em pleno desenvolvimento, o que acarreta mudanças praticamente diárias em seu código. A versão utilizada durante toda a fase de implementação foi 1.0.b4 (Beta 4). Outro ponto importante é que, por razões técnicas e restrições de cronograma, nosso sistema foi implementado e testado somente utilizando-se o interpretador Java, não o seu *Just in Time Compiler*.

1.2 Trabalhos Relacionados

Nessa seção procuramos listar alguns trabalhos relacionados que tratam do problema de recuperação de memória. Podemos citar três técnicas como sendo básicas para a

[2] Do original em inglês: Java Virtual machine

[3] *Kaffe* é marca registrada de *Transvirtual Technologies*

implementação de um sistema automático de recuperação de memória: *Contagem de Referêcias*[15], *Mark-Sweep*[27] e *Copy-Collection* [20].

A primeira técnica não consiste realmente num algoritmo, mas sim em um método direto. Essa técnica se baseia num contador em cada objeto para manter controle sobre o número de referências que chegam a cada um deles. Quando o contador atinge o valor zero, significa que o objeto não pode mais ser utilizado pelo programa e será então dealocado.

As duas técnicas seguintes adotam a estratégia de atravessar o *heap*, identificar os objetos vivos e eliminar o restante. A diferença básica entre as duas é que, *Mark-Sweep* mantém os objetos vivos em seus endereços e coloca os endereços dos objetos mortos numa lista de blocos livres, enquanto *Copy-Collection* divide o *heap* em duas áreas de memória independente, copia os objetos vivos para uma das áreas, passando a trabalhar com a mesma e deixando os objetos mortos na antiga região de memória. Descreveremos com mais detalhes esses algoritmos na Seção 2.1. Uma vasta literatura nessa área pode ser revista a partir de [22].

A seguir, destacaremos alguns trabalhos que tratam mais especificamente dos problemas relacionados ao nosso projeto: compactação de memória e *precisão*[4] da coleta na máquina virtual Java.

Bartlett, em seu trabalho *Compacting Garbage Collection with Ambigous Roots* [8], apresenta um algoritmo que ele denomina de *Mostly-Copying Collector*. Esse algoritmo trabalha num ambiente onde o coletor não tem informação precisa sobre o conjunto de raízes do *heap*. Utilizando dados extraídos dos registradores e da pilha de execução, o algoritmo divide o *heap* em dois grupos de objetos: aqueles que *podem* estar sendo referenciados por apontadores na pilha e registradores, e aqueles que certamente não estão. Os objetos do primeiro grupo permanecem em seus lugares e aqueles que pertencem ao segundo grupo são compactados através de um procedimento similar ao algortimo de *Copy-Collection*. Esse trabalho traz, em seu apêndice, um coletor implementado para trabalhar com a linguagem C. Essa abordagem é bastante similar à que implementamos em nosso sistema de compactação, uma vez que considera como fixos os objetos marcados conservativamente. Outros trabalhos nessa linha foram desenvolvidos por Boehm & Wieser [9], Appel & Hanson[5] e Boehm & Shao [10].

Também existem alguns trabalhos que foram desenvolvidos visando solucionar o problema de ambiguidade na determinação das raízes. Em se tratando especificamente da máquina virtual Java, existem os seguintes trabalhos. Agesen e Detlefs, em seu trabalho *Finding References in Java Stacks* [1], se baseiam numa abstração do método primeiramente definido por Gosling [21] para a verificação do *bytecode* Java. Esse método afirma que: se dois caminhos de execução levam a uma instrução I, que utiliza uma variavel v,

[4] Devemos entender por precisão, nesse contexto, o quão preciso o coletor consegue ser ao determinar as raízes do *heap*.

então I deve assumir que o tipo de v é a união de seus tipos ao longo dos dois caminhos. A união de dois tipos é o menor tipo geral que contenha esses dois tipos. Em Java, sempre existe tal super-tipo se os tipos unidos são tipos objetos (referências). Caso um dos tipos seja uma referência e o outro, um tipo primitivo, então não existe tipo comum. Para esses casos é definido um tipo chamado *conflito*. Todo uso de uma variável cujo tipo seja conflito é considerado ilegal. Assim, a máquina virtual é sempre capaz de decidir o tipo de uma variável que está sendo utilizada num determinado ponto do programa. Como consequência, os mapas de pilha [5] em Java não precisam considerar dependências no caminho de execução, somente em cada ponto. Existe uma exceção à essa regra: as instruções `jsr` e `ret` presentes no bytecode Java. Essas instruções são usadas na implementação de estruturas do tipo `try` {*body*} `finally` {*handler*}, sendo que `jsr` grava um endereço de retorno na pilha de operandos e salta para o endereço de uma subrotina, enquanto `ret` salta para um endereço de retorno armazenado numa variável. Nesse caso é permitido a uma variável que não seja utilizada nessa sub-rotina, conservar o tipo que possuia antes da chamada. O problema surge quando existem duas chamadas através da instrução `jsr` dentro de um mesmo método, se essa variável tiver um tipo referência antes de uma das chamadas e um tipo não referência antes da outra chamada. Se ocorrer coleta enquanto um *thread* estiver executando a sub-rotina, não existe como o coletor saber se essa variável carrega uma referência ou não. Agesen e Detlefs[1] descrevem como fazer a abstração desse método para gerar mapas de pilha, identificando quais os *slots* que possuem variáveis que são referências a objetos no *heap*, inclusive resolvendo o problema imposto pela instrução `jsr`. Para tal, são feitas alterações no bytecode gerado dependendo do tipo de conflito numa dada variável. Podem existir dois tipos de conflitos:

- **ref-uninit:** conflito de tipo referência com variável não inicializada. A solução é adicionar código no início do método para inicializar a variável com uma referência nula.
- **ref-nonref:** conflito de tipo referência com tipo que não é uma referência. A solução adotada foi dividir a variável em duas, uma com tipo referência e outra não.

Esses mesmos autores estenderam esse trabalho em *Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines* [2], onde é acrescentado o cálculo de longevidade das variáveis locais presentes na pilha, de modo a diminuir o conjunto de raízes, ou seja, em cada ponto da execução é possível saber quais referências contidas na pilha ainda são vivas, sendo estas as únicas a serem incluídas no conjunto de raízes.

[5] Do original em inglês: *stack maps*

Stichnoth *et al.* realizaram um trabalho nessa mesma linha, chamado *Support for Garbage Collection at Every Instruction in a Java Compiler* [29]. Nesse trabalho, eles afirmam que os compiladores normalmente classificam apenas algumas instruções como sendo seguras para realização de coletas de lixo [6]. Exemplos de tais instruções podem ser chamadas de sub-rotinas e desvios regressivos [7]. Os autores apresentam um método para tornar todas as instruções seguras para realização de coletas, no compilador Java. Esse fator se torna especialmente importante em aplicações *multithreads* pois, em compiladores típicos, quando uma *thread* consome toda a memória disponível e invoca o coletor, é bastante provável que as outras *threads* não estejam em pontos seguros. A solução usual para esse problema é avançar a execução das *threads* até que todas estejam em pontos seguros. O argumento frequentemente utilizado contra tornar todas as instruções seguras para coletas é o crescimento exagerado dos mapas gerados com informações necessárias ao coletor. Utilizando técnicas simples de compressão de dados, como Huffman [16], os autores conseguiram reduzir o tamanho desses mapas para até 20% do tamanho total do código gerado, um resultado que afirmam ser competitivo com os melhores existentes até o período de sua publicação. O esquema de extração das informações para o suporte ao coletor implementado se baseia em duas análises de fluxo de dados, realizadas na sua representação intermediária.

Outra contribuição desse trabalho foi apresentar uma nova solução ao problema gerado pela instrução `jsr`. Essa abordagem tem o mérito de conseguir resolver os conflitos de tipos sem necessidade de alteração do *bytecode* gerado.

1.3 Organização da Dissertação

Essa dissertação está dividida em seis capítulos, organizados da seguinte forma:

- **Capítulo 2:** Nesse capítulo apresentamos uma descrição teórica das técnicas mais utilizadas para a implementação de sistemas de recuperação de memória. Descrevemos também o sistema de gerenciamento de memória original da máquina virtual *Kaffe*.

- **Capítulo 3:** Esse capítulo contém uma descrição dos experimentos com a codificação dos endereços dos objetos vivos, usando *Binary Decision Diagrams*, que pretendíamos utilizar para extrair informações que facilitassem o processo de compactação dos dados.

[6] Normalmente referidas na literatura como *GC-Safe intructions*

[7] Do original em inglês: backward branches

- **Capítulo 4:** Nesse capítulo são descritos alguns algoritmos para compactação existentes na literatura e, em seguida, apresentamos o algoritmo implementado para o nosso sistema de recuperação de memória, destacando as diferenças com relação aos algoritmos existentes devido as restrições mencionadas acima.

- **Capítulo 5:** Esse capítulo apresenta a coleção de resultados obtidos com os testes realizados utilizando-se o pacote de benchmarks SPEC JVM98.

- **Capítulo 6:** Nesse capítulo final, apresentamos os principais problemas encontrados na implementação do projeto, apontando algumas possíveis soluções para os mesmos, encerramos com uma conclusão final e descrição de possíveis trabalhos futuros.

Capítulo 2

Algoritmos Para Recuperação Automática de Memória

2.1 Técnicas Básicas

Nessa seção descreveremos alguns algoritmos básicos para a elaboração de um sistema automático de gerenciamento dinâmico de memória.

2.1.1 Contagem de Referências

No método de *Contagem de Referências*[1] para recuperação de memória cada objeto possui um campo, que normalmente faz parte de um cabeçalho associado ao mesmo, para a contagem do número de apontadores direcionados a ele. Por exemplo, cada vez que uma nova referência é criada ou uma já existente destruída, através de uma expressão de atribuição, o compilador emite instruções para atualizar a contagem. A memória ocupada por um objeto é recuperada quando sua contagem de referências chegar a zero.

Uma das principais virtudes desse algoritmo é ser um método direto, isto é, o trabalho referente à coleta é realizado intercaladamente com a execução do programa, evitando assim grandes interrupções. Essa pode ser uma característica importante em sistemas interativos, que precisam fornecer resposta em tempo real.

Quando a contagem de referências de algum objeto chega a zero, e sua memória precisa ser recuperada, faz-se necessária a atualização dos contadores existentes em todos os objetos que são apontados por ele. Isso pode provocar a liberação de estruturas de dados inteiras se o único apontador para essa estrutura vinha exatamente do objeto morto. Essa situação está ilustrada na Figura 2.1: quando o apontador armazenado na raiz for

[1] Do original em inglês: *Reference Counting*

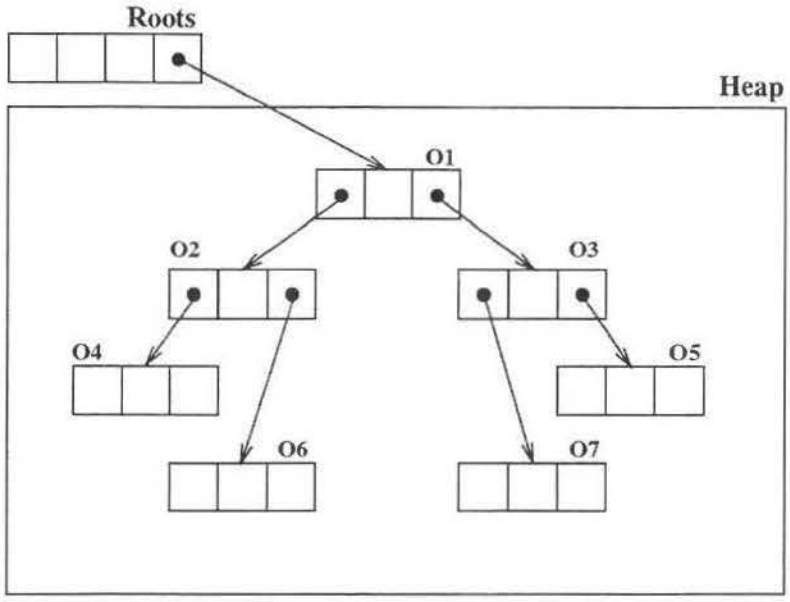


Figura 2.1: Estrutura a ser recuperada

destruído, o contador de referências de O1 será zerado, fazendo com que os contadores de O2...O7 também atinjam o valor zero, ocasionando a liberação de toda a estrutura.

Para acentuar ainda mais a característica incremental do algoritmo, o trabalho de recuperação de grandes estruturas descrito acima não precisa ser realizado todo de uma vez, pois isso acarretaria uma longa pausa na aplicação. O algoritmo pode ser implementado de maneira a distribuir esse trabalho ao longo da execução.

Outra vantagem desse método está relacionada à localidade de referência. Um objeto pode ser liberado sem que seja necessário acessar objetos em outras páginas (a menos de seus eventuais descendentes). Isso não ocorre com outros métodos que são obrigados a atravessar todos os objetos vivos do *heap* antes de liberar qualquer objeto morto.

Por outro lado, o método de contagem de referências possui sérias desvantagens. Podemos destacar como as mais críticas o alto custo de processamento para manter atualizados os contadores de referências e a impossibilidade de recuperar a memória utilizada por estruturas de dados cíclicas.

Cada vez que um apontador é criado ou destruído um contador de referências precisa ser atualizado. Quando um apontador for sobrescrito, dois contadores terão que ser alterados: o do antigo objeto alvo será decrementado e o do novo objeto alvo, incrementado. Além disso, a cada decremento num contador é necessário verificar se ele atingiu o valor zero. Desta maneira, podemos perceber que o custo de todas as instruções emitidas para a realização desse controle tende a se tornar muito alto.

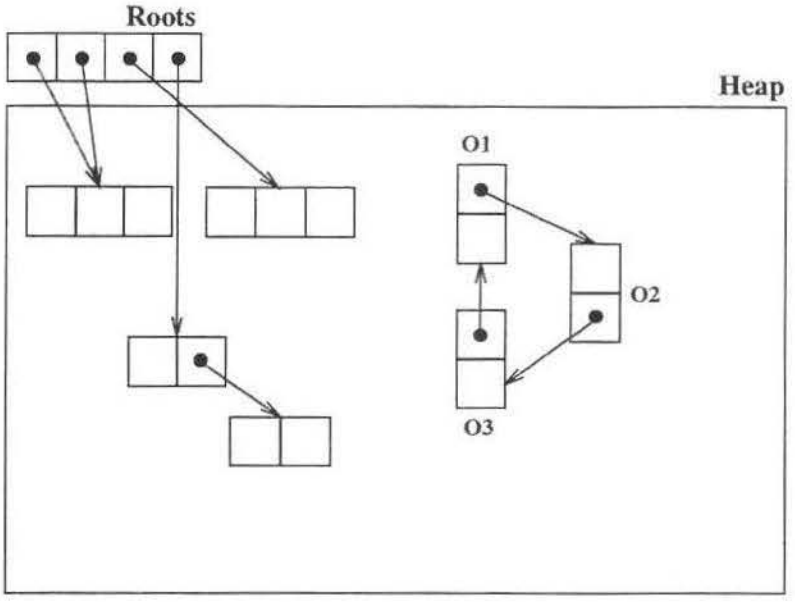


Figura 2.2: Estrutura Cíclica

Existem técnicas para amenizar esse custo. Uma delas é a utilização de *Análise de Fluxo de Dados*. Através dessa análise o compilador consegue juntar sequências de incrementos ou decrementos num contador em uma única instrução, ou até mesmo eliminá-los se a variação final no valor do contador for zero. Outra técnica, conhecida como *Deferred Reference Counting*[31], não atualiza imediatamente os contadores com referências vindas de variáveis locais. Isso elimina o esforço inútil causado pela ativação de procedimentos que retornam logo após serem chamados, cujos argumentos causariam incrementos e decrementos sucessivos. Nesse caso, os contadores são ajustados somente para refletir as referências entre objetos do *heap* e, de tempos em tempos, são atualizados a partir dos apontadores da pilha para que os objetos com contadores nulos possam ser seguramente considerados mortos e, consequentemente, liberados.

A presença de estruturas de dados cíclicas se torna um problema porque os contadores de referências de seus objetos componentes nunca atingem o valor zero. Observamos no exemplo dado pela Figura 2.2 que, apesar da estrutura formada por `O1...O3` não ser alcançável por nenhum caminho através das raízes, cada objeto ainda possui um apontador chegando até ele. Desse modo, os contadores possuem valor 1 e os objetos não serão liberados.

Nesse caso existem dois tipos de solução. A primeira é obrigar o programador a romper a circularidade deste tipo de estrutura quando estas se tornarem inúteis, o que vai de encontro com a idéia de livrá-lo de qualquer preocupação quanto à liberação de

memória. A segunda é intercalar coletas que utilizem outro algoritmo que não possua esse problema, como *Mark-Sweep*, garantindo assim a liberação da memória consumida por essas estruturas, mesmo que não seja imediatamente. Alguns trabalhos que sugerem soluções para o problema de contagem de referências com estruturas cíclicas nessa linha de métodos híbridos são [26] e [25].

Em vista dessas desvantagens que geralmente o tornam menos eficiente que coletores baseados em *Mark-Sweep* ou *Copy-Collection*, o método de contagem de referências acaba sendo raramente utilizado em ambientes de linguagem de programação.

2.1.2 Mark-Sweep

Esse algoritmo, como sugere o próprio nome, é composto de duas fases:

Marcação: A primeira fase utiliza a abordagem de atravessar o *heap*, a partir de suas raízes, para identificar (marcar) os objetos vivos. Normalmente usa-se os algoritmos de busca em largura ou profundidade para percorrer o *heap*.

Varredura: O algoritmo atravessa todo o *heap*, liberando os objetos que não foram marcados, ou seja, que não são alcançáveis a partir das raízes e que, portanto, estão mortos. Os objetos liberados são colocados numa lista de blocos livres para posterior reutilização por parte do alocador.

Mark-Sweep não é um algoritmo naturalmente incremental, pois a aplicação tem sua execução interrompida enquanto uma coleta está sendo realizada. Os objetos mortos não são recuperados assim que se tornam inúteis, como no método de contagem de referências, mas permanecem no *heap* até que o coletor seja disparado. Isto geralmente ocorre quando não há mais memória disponível para satisfazer uma requisição do alocador. Esta característica torna esse algoritmo, em sua forma tradicional, praticamente inviável para aplicações de tempo real. Existem técnicas mais avançadas, como *Generational Collection* e *Incremental Collection* (ver seção 2.2), que podem ser aplicadas em conjunto com *Mark-Sweep* visando diminuir o tempo de latência do coletor e, consequentemente, seu impacto no desempenho da aplicação.

Existem duas grandes vantagens de *Mark-Sweep* sobre contagem de referências. A primeira é com relação a estruturas de dados cíclicas, que são recuperadas naturalmente nesse algoritmo, desde que não haja caminho para nenhum de seus objetos a partir das raízes. A segunda diz respeito ao acréscimo de processamento devido à atualização dos contadores, no algoritmo de contagem de referências. Esse acréscimo deixa de existir em algoritmos que atravessam o *heap* para identificar os objetos vivos, como *Mark-Sweep*.

Como dito anteriormente, utiliza-se um algoritmo de busca para a implementação da primeira fase do algoritmo *Mark-Sweep*. Algoritmos de busca normalmente são implementados de forma recursiva, mas para a utilização em coleta de lixo isto se torna praticamente inviável. Isto ocorre devido ao grande espaço de memória requerido para a pilha, resultante muitas vezes da grande quantidade de objetos que podem estar alocados no *heap*. Isto pode levar à ocorrência de *overflows* da pilha.

Uma solução que pode ser adotada para esse tipo de problema é a implementação do algoritmo substituindo-se as chamadas recursivas por laços iterativos, economizando assim o tempo gasto nas chamadas e retornos e também o espaço dos registros de ativação. Nesse caso faz-se necessária a presença de uma estrutura de dados auxiliar para armazenar os ponteiros para objetos que já foram identificados como vivos mas que ainda não tiveram os descendentes visitados. O maior caminho possível dentro do grafo é um limitante superior para a profundidade dessa recursão, significando que com a adoção de um pilha explícita teríamos um custo de no máximo T palavras num *heap* de tamanho T. Essa ainda é uma perda considerável.

Outra solução possível é conhecida como *Pointer Reversal*, que é atribuída por Knuth em [23] a Peter Deutsch, Herbert Schorr e W.M. Waite. A idéia é usar campos dos registros já atravessados para armazenar implicitamente uma pilha. Suponha, por exemplo, que o registro y está sendo visitado, que o próximo registro a ser visitado é apontado pelo campo $y.f_i$ de y, e que o último registro visitado antes de y (e.g. x) está armazenado em uma variável temporária t. Assuma também que $y.f_i$ aponta para um registro z. Neste caso, imediatamente antes de z ser visitado o campo $y.f_i$ pode ser trocado pela variável temporária t. Ou seja, agora t aponta para z, e $y.f_i$ aponta para x. Deste modo, o caminho de retorno do algoritmo DFS está armazenado nos campos $y.f_i$ dos registros atravessados pelo mesmo até aquele ponto de sua execução. Os valores originais dos campos $y.f_i$ podem ser facilmente recuperados uma vez que, durante o retorno do algoritmo DFS, este contém um apontador para o registro de onde está retornando.

O desempenho de um coletor baseado em *Mark-Sweep* pode cair consideravelmente dependo da quantidade de espaço no *heap* que é ocupada pela aplicação. Se mesmo após uma coleta restarem muitos objetos no *heap*, isso implica que o coletor recuperou uma quantidade pequena de memória. Isso se repetindo com frequência pode ocasionar um número excessivo de chamadas ao coletor, acarretando um grande atraso na execução do programa. Esse problema não existe no esquema de contagem de referências, por não precisar interromper a aplicação para a recuperação de memória.

A *fragmentação* de memória também pode ser considerada uma desvantagem desse algoritmo, pois os objetos vivos ficam intercalados aos espaços livres de memória. Esses blocos livres normalmente fazem parte de uma lista, para que o alocador possa reaproveitá-los. Com a presença de objetos de tamanhos diferentes podemos chegar a uma situação

onde a quantidade total de memória livre seja suficiente para alocar um novo objeto, mas essa memória esta dividida em pequenos blocos não contíguos, impossibilitando assim a sua utilização por um único objeto. Esse é um problema que também pode acontecer utilizando-se contagem de referências.

Exitem esquemas que visam agilizar o processo de busca de blocos livres nas listas, como criar um vetor onde cada elemento é uma lista de blocos de determinado tamanho, tentando assim tornar o desempenho compatível com o dos coletores que realizam compactação de memória. Discutiremos esses esquemas com mais detalhes quando apresentarmos o coletor da máquina virtual Java *Kaffe*, na Seção 2.3.

2.1.3 Copy-Collection

Copy Collection é mais um algoritmo que atravessa o *heap* em busca dos objetos vivos. Sendo assim, também não impõe custos adicionais nas operações de alteração de apontadores e na recuperação de estruturas cíclicas.

Esse algoritmo divide o *heap* em dois semi-espaços chamados *from-space* e *to-space*. Os dados atuais do programa estão armazenados no *to-space*, e o primeiro passo do algoritmo é inverter os papéis dos semi-espaços. A partir daí, o algoritmo atravessa os objetos vivos, contidos no *from-space*, criando uma cópia de cada um no *to-space*. Após todos os objetos terem sido visitados e copiados, os objetos mortos são abandonados no *from-space* e a execução do programa recomeça. Novos objetos necessários serão alocados no *to-space* e, quando não houver mais área livre nesse semi-espaço para atender a uma requisição do alocador, uma nova coleta será iniciada.

Desde que Cheney [14] descobriu uma técnica iterativa eficiente para a implementação de *Copy-Collection*, esse algoritmo vem sendo amplamente utilizado, inclusive como base para a implementação de técnicas avançadas como *Generational* e *Incremental Collection*.

Essa técnica possui algumas vantagens sobre outros esquemas de coleta de lixo. Seu custo é proporcional ao número de objetos vivos, e não ao tamanho total do *heap*, tornando-o especialmente interessante para aplicações onde apenas uma pequena porção do *heap* sobrevive por longo tempo.

Um efeito colateral imediato e uma das grandes vantagens de *Copy-Collection* é o fato de que, ao término de uma coleta, todos os objetos vivos estão compactados em um dos extremos de um dos semi-espaços do *heap*. Isso faz com que não exista o problema de fragmentação de memória e simplifica muito o trabalho do alocador. O processo de alocação torna-se um simples incremento no apontador para a primeira posição livre do *heap*, sendo que a presença de objetos de tamanho variável não causa dificuldades como num esquema baseado em listas de blocos livres, onde se faz necessária uma busca por um bloco que seja de tamanho suficiente para conter o novo objeto.

A maior desvantagem dos coletores baseados nesse algoritmo é a necessidade de dividir o *heap* ao meio. Isso causa uma drástica diminuição no desempenho do coletor quando o espaço ocupado pela aplicação cresce. Quanto mais objetos sobreviverem às coletas menos espaço é recuperado, ocasionando coletas mais frequentes e assim, prejudicando o desempenho da aplicação. Esse problema também atinge os coletores baseados em *Mark-Sweep*, mas a situação atinge um ponto crítico com o dobro da velocidade em *Copy-Collection*, devido ao fato do *heap* estar dividido pela metade. Apresentaremos algums dados sobre as complexidades dessas técnicas básicas para coleta de lixo na Seção 2.1.5.

Apesar da complexidade desse algoritmo ser proporcional ao número de objetos vivos, as suas constantes de proporcionalidade tendem a ser grandes. Podemos citar como exemplo o fato de que a cópia de um objeto tem um custo muito maior que apenas setar um bit para marcá-lo. Isto se agrava se a aplicação estiver trabalhando com objetos grandes. Por razões como essa um coletor baseado em *Mark-Sweep* pode vir a ser mais eficiente, apesar do custo assintoticamente maior, dependendo das características da aplicação.

A implementação de *Copy Collection* criada por Cheney [14] é baseada em dois algoritmos: o primeiro realiza a busca por registros vivos utilizando BFS (ver Algoritmo 1) e o segundo implementa a função *forward* (ver Algoritmo 2).

Nesse algoritmo, `scan` aponta para o primeiro registro no *to-space* cujos filhos ainda não foram visitados e `next` aponta para a primeira posição livre do *heap* (ver Figura 2.3). Primeiramente o algoritmo copia todas as raízes do programa para o *to-space*. Em seguida ele aplica a função *Forward()* enquanto existirem registros no *to-space* cujos filhos ainda não foram visitados, isto é, enquanto `scan` não alcançar `next`.

A operação de *forwarding* consiste em fazer com que um ponteiro para um registro no *from-space* passe a apontar para sua cópia no *to-space*. No momento da aplicação dessa função a um ponteiro `p` (ver Algoritmo 2), três situações podem ocorrer:

1. `p` aponta para um registro no *from-space* que ainda não foi copiado. Então esse registro é copiado para o *to-space* na posição apontada por `next` e um ponteiro, contendo o novo endereço do objeto no *to-space*, é criado pela função *forward* e armazenado no primeiro campo de `p` ($\texttt{p}.f_1$).

2. `p` aponta para um registro no from-space que já foi copiado. Então o campo $\texttt{p}.f_1$ possui um ponteiro para a cópia do registro apontado por `p` no *to-space*. Esse ponteiro é retornado pela função *forward*.

3. `p` não é um ponteiro, ou aponta para um registro fora do from-space. Então a função *forward* não retorna nada.

A Figura 2.3 ilustra um exemplo de funcionamento do algoritmo de Cheney

A Figura 2.3 (a) representa o início de um ciclo do *Garbage Collector*, onde as raízes apontam para registros no *from-space* e o *to-space* está vazio.

A Figura 2.3 (b) retrata o momento após as raízes terem sido copiadas para o *to-space* pela função *Forward()*. Observe que nesse momento os ponteiros guardados nos campos das raízes apontam para registros no *to-space* e os antigos registros do *from-space* que eram apontados pelas raízes, agora guardam apontadores para as suas respectivas cópias no *to-space*. Note também que os registros no *to-space* estão entre `scan` e `next`, pois seus campos ainda não foram percorridos.

Na Figura 2.3 (c), um dos registros do *to-space* (aquele que possui valor 15) já foi percorrido, então podemos observar que os registros apontados por seus campos foram copiados para o *to-space* e que o ponteiro `scan` está indicando o registro seguinte como próximo a ser percorrido.

O algoritmo seguirá até que os ponteiros `scan` e `next` estejam juntos, indicando que todos os registros alcançáveis do *from-space* foram visitados.

Algoritmo 1 Algoritmo de *Copy-collection* baseado em BFS

scan ← next ← beginning of to-space
for each root r

    $r \leftarrow$ Foward(r)

while scan < next

    for each field f_i of record at scan

        scan.$f_i \leftarrow$ Foward(scan.f_i)

    scan ← scan + size of record at scan

2.1.4 Mark-Compact

Uma das grandes desvantagens do algoritmo *Mark-Sweep* é a sua tendência em fragmentar o *heap* sob a presença de objetos de tamanhos variados. Fragmentação pode significar um maior custo de alocação, podendo inclusive tornar impossível a inclusão de um objeto grande, sem expansão do *heap*, embora a quantidade de memória disponível seja suficiente.

Coletores baseados no esquema conhecido como *Mark-Compact* vêm justamente sanar esse problema presente nos esquemas baseados em *Mark-Sweep*.

Algoritmo 2 Algoritmo da Função Forward

function Forward(p)

if p points to from-space

then if $p.f_1$ points to to-space

then return $p.f_1$ **else**

for each field f_i of p

next.$f_i \leftarrow p.f_i$

$p.f_i \leftarrow$ next next $\leftarrow$ next + size of record p **return** $p.f_i$

else return p

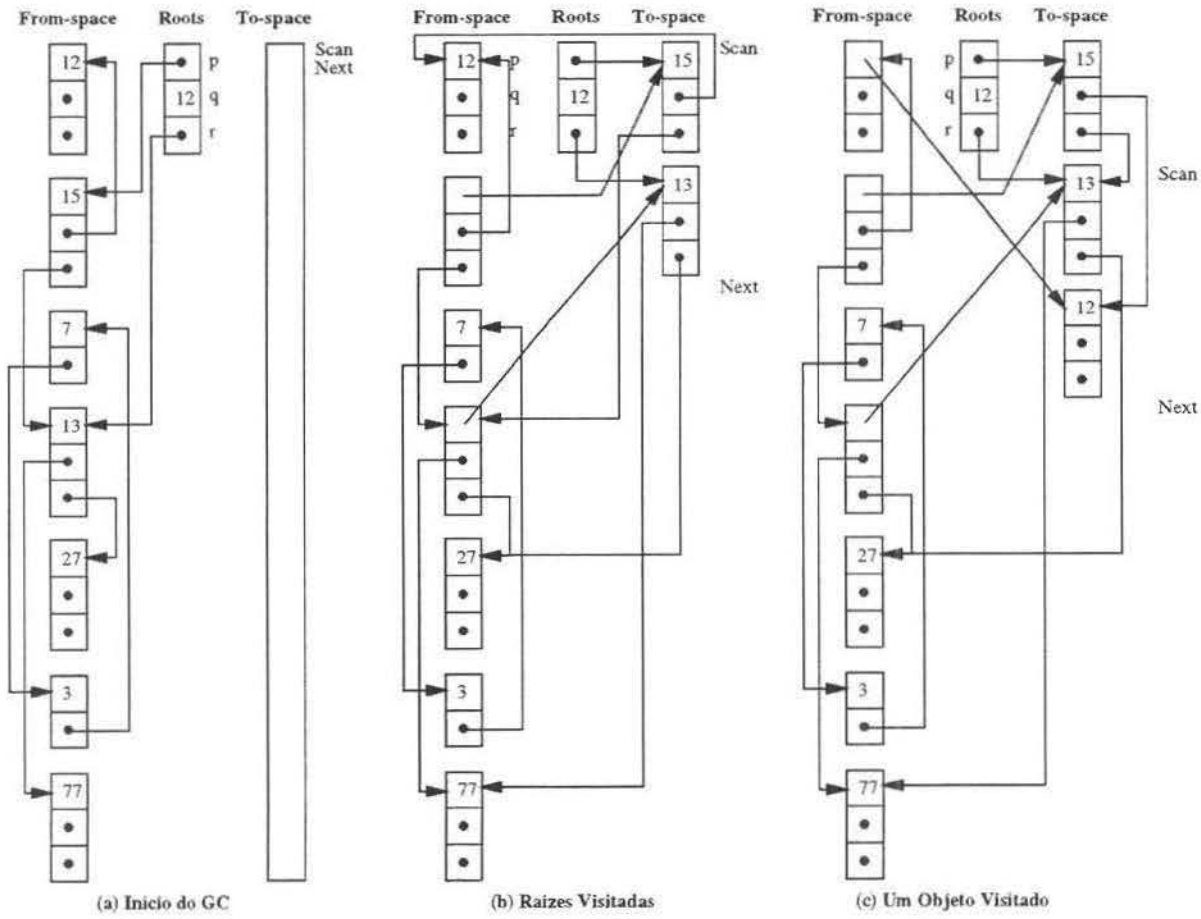


Figura 2.3: Exemplo de Copy-Collection

A primeira fase desse algoritmo é exatamente a mesma que em *Mark-Sweep*: percorrer o *heap* a partir das raízes, identificando todos os objetos vivos.

Na segunda fase, enquanto atravessa todo o *heap*, os objetos mortos não são colocados em listas de blocos livres. Ao invéz disso, o algoritmo vai compactando os objetos vivos em um dos extremos do *heap*, eliminando assim os buracos que seriam deixados entre os mesmos. Isso resolve o problema de fragmentação e torna a alocação tão simples quanto nos coletores baseados em *Copy-Collection*.

Mark-Compact ainda compartilha com *Mark-Sweep* a propriedade de ter custo proporcional ao tamanho do *heap*, sendo que podem ser necessários vários passos ao longo dos dados durante a compactação, apresentando assim uma desvantagem em relação a *Copy-Collection*. Pode-se tirar proveito no aspecto de localidade, pois a ordem de alocação pode ser muito parecida com a subsequente ordem de acesso dos objetos, e *Mark-Compact* permite que eles sejam compactados respeitando-se essa ordem.

Técnicas de como se realizar a compactação nesse esquema serão discutidas mais adiante no Capítulo 4, quando formos apresentar nosso coletor cuja fase de compactação foi baseada na abordagem *Mark-Compact*.

2.1.5 Complexidade das Técnicas Básicas

Nesta seção, ilustraremos o custo envolvido na realização da recuperação automática de memória considerando as três técnicas básicas apresentadas. Utilizaremos uma análise baseada na encontrada em [4].

Em Contagem de Referências, uma grande parcela do custo está concentrado nas instruções extras emitidas pelo compilador, afim de atualizar os contadores nos objetos. Tomemos como exemplo uma instrução de atribuição $x.f_i \leftarrow p$. Nesse método, essa instrução seria acrescentada de instruções de controle, o que resultaria numa sequência como a apresentada abaixo:

$$
\begin{array}{ll}
z & \leftarrow x.f_i \\
c & \leftarrow z.count \\
c & \leftarrow c-1 \\
z.count & \leftarrow c \\
if\ c = 0 & \text{goto } PutOnFreelist \\
x.f_i & \leftarrow p \\
c & \leftarrow p.count \\
c & \leftarrow c+1 \\
p.count & \leftarrow c
\end{array}
$$

Como vimos na Seção 2.1.1, existem técnicas que visam diminuir a quantidade de pontos do programa onde são necessárias essas instruções extras de controle, pois sua presença em conjunto com toda instrução de atribuição acarretaria num custo altíssimo. Entretanto, muitas dessas instruções de incremento e decremento dos contadores permanecem. Esse alto custo e o problemas com ciclos fazem com que, na prática, esse método não seja o mais utilizado em sistemas automáticos de gerenciamento de memória [4].

Em se tratando dos métodos que atravessam o *heap*, *Mark-Sweep* e *Copy-Collection*, o custo relacionado a cada instrução de atribuição não existe. O custo está concentrado nos algoritmos utilizados para identificar os objetos vivos e liberar a memória utilizada pelos objetos mortos. Para ilustrar esse custo nessas duas técnicas, vamos supor que M seja a quantidade total de memória no *heap* e V seja a quantidade total de memória gasta pelos objetos vivos.

O algoritmo *Mark-Sweep* atravessa o grafo de objetos vivos, através de apontadores entre eles, durante sua fase de marcação. Na sua segunda fase, a varredura, *Mark-Sweep* atravessa todo o *heap*, liberando objetos não marcados. O custos de marcar (c_1) ou liberar(c_2) um objeto podem ser considerados constantes. A quantidade de memória recuperada nesse método é de $M-V$. Logo, podemos aproximar a complexidade temporal, amortizada pela quantidade de memória recuperada, nesse algoritmo por:

$$C_{MS} = \frac{c_1.V+c_2.M}{M-V},$$

onde c_1 representa o custo de cópia de um objeto.

Copy-Collection atravessa somente os objetos vivos porém, trabalha utilizando metade do *heap* como espaço de alocação. Assim, analogamente a *Mark-Sweep*, a complexidade de *Copy-Collection* pode ser aproximada por:

$$C_{CC} = \frac{c_3.V}{M/2-V}$$

Note que, se considerarmos apenas a parte temporal das complexidades acima (numerador das frações), a complexidade de *Copy-Collection* é menor que a de *Mark-Sweep*. Mas, quando levamos em consideração a ocupação de memória (denominador das frações), vemos que *Mark-Sweep* apresentará problemas de eficiência quando V se aproximar de M, sendo que *Copy-Collection* sofrerá o mesmo problema quando V se aproximar de $M/2$. Daí vem a afirmação de que *Copy-Collection* apresenta problemas de desempenho com relação à ocupação do *heap* com o dobro da velocidade de *Mark-Sweep*.

Mais detalhes sobre as complexidades dos algoritmos podem ser obtidos em [30] e [22].

2.2 Técnicas Avançadas

Nessa seção serão apresentadas duas técnicas elaboradas que, trabalhando em conjunto com uma das técnicas básicas, usualmente melhoram o desempenho do coletor.

2.2.1 Generational Collection

Existem estudos na literatura que mostram claramente que a maioria dos objetos no *heap* permanece viva por um período muito curto de tempo. Isto acaba fazendo com que a grande maioria dos objetos alocados logo após uma coleta não chegue viva até a próxima. Observou-se mais ainda, os poucos objetos que sobrevivem a uma coleta tendem a sobreviver por muito mais tempo. Em seu *survey* sobre *garbage collection*, Paul R. Wilson [31] afirma que de 80 a 98% dos objetos normalmente morrem antes que o próximo *megabyte* seja alocado. Nesse caso, uma medida frequente para a idade do objeto é a quantidade de memória alocada desde sua criação. Isto devido a dois fatores: é uma medida independente de máquina, e a quantidade de memória alocada está intimamente ligada à frequência das coletas.

Generational Collection é uma técnica criada para tirar proveito dessas características de longevidade dos objetos. A proposta aqui é dividir o *heap* em dois ou mais semi-espaços, chamados gerações, de acordo com a idade dos objetos que as mesmas contém. As gerações de objetos mais jovens são coletadas frequentemente pois seus objetos têm uma maior probabilidade de estarem mortos e, portanto, de serem liberados. As gerações de objetos antigos são coletadas com muito menos frequência pois é muito menos provável que uma boa quantidade de memória seja recuperada a partir delas.

Isso pode trazer um grande ganho em relação às pausas impostas na aplicação, pois as coletas serão realizadas em áreas reduzidas do *heap*, tornando-as muito mais rápidas. O trabalho de percorrer ou copiar os mesmos objetos por várias coletas seguidas é eliminado, pois estes pertencem a gerações mais antigas, cujas coletas podem ser retardadas até pontos onde a aplicação está realizando um grande volume de processamento e, portanto, o usuário não deve estar esperando respostas imediatas.

A aplicação desse esquema de coleta nos obriga a ter um cuidado especial com ponteiros de objetos pertencentes a gerações mais antigas para objetos em gerações mais jovens. Isto porque eles não seriam percorridos, a partir das raízes, numa coleta que somente abrangesse as gerações jovens, podendo ocasionar a liberação errada de algum objeto alcançável a partir de objetos mais velhos.

Podemos ilustrar tal situação através da Figura 2.4, onde o objeto `E` pertencente a geração mais antiga possui um apontador para o objeto `d` (referência tracejada na Figura), que faz parte da geração mais jovem. Note que, se somente percorressemos os apontadores a partir das raízes para objetos na geração jovem (espaço `Young`), não alcançaríamos o objeto `d`. Esse objeto seria dado como morto, causando sua incorreta dealocação, quando na realidade ainda é vivo, pois pode ser alcançado a partir do objeto `E`, em `Old`.

Para que não cause problemas, esse tipo de apontador deve ser considerado como raiz para uma eventual coleta que abranja essa geração mais jovem. Esses apontadores são criados a partir de duas situações: quando um objeto é promovido a uma geração

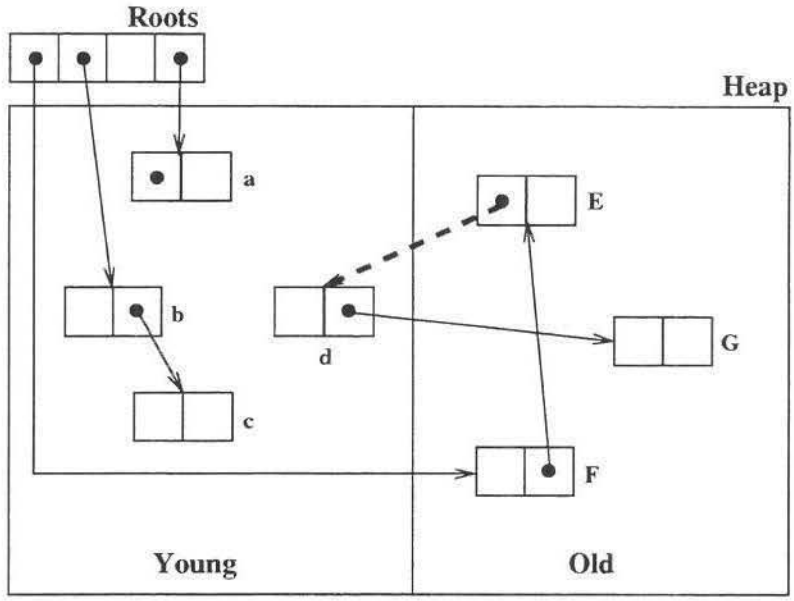


Figura 2.4: Apontador Entre Gerações

mais antiga após sobreviver a um determinado número de coletas ou quando o ponteiro é explicitamente armazenado pela aplicação.

O primeiro caso pode ser facilmente identificado e catalogado pelo coletor. Para aqueles criados no segundo caso, se faz necessário um esquema do tipo *write barrier* para detectar o apontador no momento em que é criado e armazená-lo para ser percorrido posteriormente como se fosse uma raiz. Felizmente não são todos os ponteiros que precisam ser armazenados, podemos descartar aqueles armazenados em variáveis locais do programa, uma vez que elas sempre fazem parte do conjunto de raízes e portanto são visitadas de qualquer maneira. Isso faz com que esse processamento extra não cause grande impacto na aplicação. Além disso, precisamos garantir que as gerações mais jovens serão incluídas nas coletas que abrangem as gerações antigas, para assegurar que esse é realmente o único tipo de apontador que requer tratamento especial.

O objetivo de *Generational Collection* é tirar vantagem de uma hipótese que é tida como válida para a maioria das aplicações. Certamente existem casos onde o uso de *Generational Collection* pode prejudicar o desempenho. Porém, no caso geral, o seu uso em conjunto com *Mark-Sweep* ou *Copy Collection* é tido como uma boa maneira de melhorar o desempenho dessas duas técnicas básicas, principalmente com relação a pausas na execução.

2.2.2 Incremental Collection

Na seção anterior foi apresentada uma técnica para a redução do tamanho das pausas impostas pelo coletor: *Generational Collection*. Destacamos que, com a utilização de tal técnica, é esperada uma redução na duração das pausas para coleta de lixo mas que, dependendo de como os dados se comportam, o pior caso pode trazer resultados muito ruins [31].

Esse pode ser um risco inaceitável para aplicações interativas que demandam respostas em tempo real e não toleram atrasos. Em tais casos, se faz necessária uma técnica que reduza o tempo das pausas na execução, mas que também garanta respostas rápidas no pior caso. Para isso, a coleta não pode mais ser tratada como uma operação atômica, isto é, a aplicação sendo interrompida até que todo o ciclo de uma coleta seja completado. Ao invéz disso, é necessário que as operações realizadas pelo coletor sejam intercaladas às instruções executadas pela aplicação. A técnica que realiza tal abordagem é conhecida como *Incremental Collection*[31].

Vimos que contagem de referências é um algoritmo considerado naturalmente incremental. Isso porque o trabalho de coleta, a menos da liberação recursiva de vários objetos, está todo distribuído ao longo da execução do programa. Veremos agora que coletores baseados em *Mark-Sweep* e *Copy-Collection* também podem se tornar incrementais.

O principal problema que surge com a aplicação dessa abordagem é manter a coerência da estrutura de dados do *heap* entre a aplicação e o coletor. Uma vez que a coleta vai estar intercalada à execução do programa, este pode fazer modificações no grafo após o início de um ciclo de coleta e, de alguma maneira, o coletor precisa ser *avisado* sobre essas mudanças. Pelo fato da aplicação realizar *mutações* no grafo durante um ciclo de coleta, ela é chamada de *mutator* quando tratada num contexto incremental.

Isso trás à tona outro aspecto a ser considerado: o grau de conservadorismo com que serão tratadas as mudanças feitas pelo *mutator* durante uma coleta. Essas mudanças podem fazer com que objetos que estavam vivos no início de uma coleta se tornem mortos durante esse mesmo ciclo. Nos coletores mais conservativos esses objetos só serão liberados numa próxima coleta, onde já estarão mortos desde o início, gerando assim o chamado lixo flutuante.

Problemas de coerência surgem quando temos vários processos que compartilham dados que podem estar sendo alterados. Coletores incrementais baseados em *Mark-Sweep* possuem um tipo de problema conhecido como *multiple readers, single writer*: o coletor precisa responder às mudanças, mas somente o *mutator* pode alterar o grafo de objetos. Da mesma maneira, somente o coletor pode alterar o campo que contém os bits de marcação. Sendo assim, campos contendo apontadores só podem ser alterados pelo *mutator* e os campos de controle da coleta só podem ser alterados pelo coletor.

Em se tratando de coletores baseados em *Copy-Collection*, surge um outro tipo de

problema de coerência, conhecido como *multiple readers, multiple writers*: tanto o coletor quanto o *mutator* podem alterar o grafo de objetos, e ambos precisam saber como tratar as mudanças feitas pelo outro.

Felizmente, com respeito a recuperação de memória, esses problemas podem ser atacados de uma forma relaxada, isto porque não se faz necessária a exigência de que os dois lados, *mutator* e coletor, tenham uma visão exatamente igual dos dados do *heap*, desde que as diferenças não causem erros na semântica do algoritmo. O coletor pode vir a enxergar alguns objetos que são mortos como estando vivos. Essa situação ocorre quando um objeto torna-se lixo após o coletor já tê-lo visitado. É preciso garantir que ele não veja algum objeto vivo como estando morto, o que levaria a uma liberação errada.

Quanto mais relaxado for o tratamento da consistência entre o coletor e o *mutator*, maior a quantidade de lixo flutuante gerada. Em contrapartida, maior será a liberdade na implementação do algoritmo de coleta.

Em *Incremental Collection* é muito usada uma abstração introduzida por Dijskstra [19], chamada de *Marcação por Três Cores*. Esse esquema se baseia em três listas (brancos, cinzas e pretos) onde:

Brancos: são os objetos que o coletor ainda não alcançou. Ao final da fase de marcação, indicam os objetos que estão mortos.

Cinzas: são os objetos que foram marcados pelo coletor mas cujos campos ainda não foram percorridos ou, em um esquema incremental, pode indicar também que o *mutator* alterou a conectividade desse objeto com o resto do grafo, forçando o coletor a visitá-lo novamente.

Pretos: indica que o objeto já foi visitado assim como seus descendentes imediatos. O coletor não precisará mais visitá-lo.

Na Figura 2.5, o objeto `A` já foi percorrido pelo coletor, e não possui descendentes a serem percorridos. Isto significa que o coletor não passará mais por `A`, então ele foi colorido com a cor preta. Os objetos `B` e `D`, já foram visitados a partir dos apontadores vindos das raízes, mas seus descendentes ainda não foram atingidos pelo coletor. Isso faz com que possuam a cor cinza. Já os objetos `C`, `E` e `F` ainda não foram atingidos pelo coletor, por isso ainda possuem a cor branca. Ao final dessa coleta, os objetos de `A` a `E` serão atingidos pelo coletor, terão seus descendentes percorridos, portanto, todos estarão coloridos pela cor preta. No caso de `F`, ele não pode ser atingido por nenhum caminho a partir da raiz, por isso permanecerá com a cor branca durante todo o processo e, ao final da coleta, será dado como morto.

Durante o processo de uma coleta, os objetos vivos passam de brancos para cinzas e posteriormente para pretos. Ao longo desse período é preciso manter invariante a seguinte

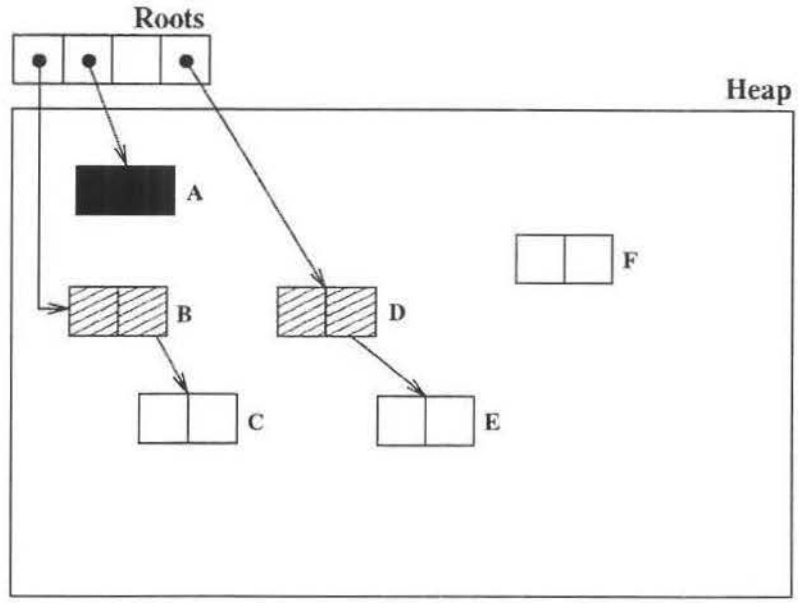


Figura 2.5: Exemplo de Marcação por três cores

característica: não existem objetos pretos apontando para objetos brancos. Isso é muito importante para garantir a corretude do algoritmo. Como vimos, um objeto possuir a cor preta significa, para o coletor, que ele já foi visitado e que todos os seus apontadores para descendentes também já foram percorridos. Isso implica que os filhos desse objeto devem possuir a cor cinza ou preta, por já terem sido visitados pelo menos uma vez pelo coletor. Mais ainda, significa que o coletor não percorrerá novamente qualquer dos apontadores contidos no objeto preto. Então, a existência de um objeto branco sendo apontado por um objeto preto pode fazer com que o primeiro não seja alcançado pelo coletor, por consequência seja considerado morto, provocando uma falha no algoritmo. Desse modo, se fazem necessárias medidas para prevenir que mudanças no grafo de objetos, realizadas pelo *mutator*, introduzam apontadores de objetos pretos para objetos brancos.

Uma das abordagens para esse problema é não permitir que o *mutator* acesse objetos brancos. Toda vez que o *mutator* tentar acessar um ponteiro para um objeto branco, o coletor é acionado e imediatamente transfere esse objeto para a lista de cinzas, garantindo assim que será percorrido nessa coleta. Essa operação é conhecida como *read barrier*.

A outra abordagem é a utilização de operações conhecidas como *write barrier*. Nesse caso, quando o *mutator* escreve um apontador para um objeto branco, esse apontador é armazenado pelo coletor de forma que ele possa ser percorrido posteriormente.

Para que o coletor libere erroneamente um objeto vivo devido ao problema descrito acima, dois fatos precisam ocorrer concomitantemente:

1. O *mutator* precisa escrever um apontador de um objeto preto para um objeto branco;

2. A referência original do objeto precisa ser destruída, fazendo com que a referência criada acima seja a única existente.

Se a condição 1 não for satisfeita, nada precisa ser feito porque as referências a partir de objetos cinzas serão percorridas pelo coletor e, se nem essas referências existirem, então o objeto está morto e deve ser liberado.

Se a segunda condição não for atendida significa que o objeto ainda é alcançável através de sua referência original e, portanto, será mantido pelo coletor.

Os métodos de *write barrier* resolvem esse problema de consistência entre o *mutator* e o coletor prevenindo-se para que uma das condições acima não aconteçam.

O coletor implementado por Baker [7] é, provavelemente, o mais conhecido entre os algoritmos incrementais. Esse trabalho é baseado no algoritmo de *Copy Collection*, descrito na Seção 2.1.3, e utiliza a técnica de *read-barrier* para coordenação com o *mutator*. O algoritmo original de Baker realiza a cópia dos objetos da mesma maneira que o algoritmo de Cheney, mas seu trabalho é intercalado com as operações realizadas pelo *mutator*. O ciclo de uma coleta se inicia com uma operação atômica: A inversão dos semi-espaços e cópia dos objetos apontados pelas raízes para o *to-space*. A partir desse ponto o *mutator* pode voltar a tomar controle da execução. A operação inicial torna os objetos deixados no *from-space* conceitualmente inválidos. Com isso, todo objeto pertencente a esse semi-espaço que seja acessado pelo *mutator*, precisa ser primeiramente copiado para o *to-space*. Tal cópia é forçada através de uma operação de *read barrier*, que normalmente é implementada através da emissão, pelo compilador, de instruções a serem executadas a cada acesso a um apontador. Esse sistema adota uma abordagem conservativa para garantir a consistência do *heap*: todo objeto alocado durante um ciclo de coleta é assumido como estando vivo, em termos do esquema de três cores para marcação, todo novo objeto alocado terá a cor preta. Isso implica que um objeto que seja alocado e morra durante uma mesma coleta, só será recuperado no próximo ciclo.

Descrições detalhadas dos métodos de *write barrier* bem como algoritmos que implementam a técnica de *Incremental Collection* podem ser encontrados em [22] e [31].

2.3 Garbage Collection na Máquina Virtual *Kaffe*

O sistema de gerenciamento de memória da máquina virtual *Kaffe* [28] é baseado no algoritmo de *Mark-Sweep*. Ele utiliza o esquema de marcação por três cores, descrito na Seção 2.2.2. A principal razão para a introdução desse esquema no coletor é a intenção

de seus desenvolvedores de torná-lo incremental. Apesar disso, até o presente momento (*circa* 1999) esse coletor ainda é completamente sequencial.

Na *Kaffe*, o *heap* é dividido em páginas e objetos grandes e pequenos são tratados de maneira diferenciada. Os objetos pequenos são alocados em páginas onde todos os objetos são de mesmo tamanho. Cada uma dessas páginas possui, em seu início, um bloco que contém as informações necessárias para localizar o objeto dentro da página e determinar o tratamento que o mesmo deve sofrer do *Garbage Collector*, isto é, indica se o objeto precisa ser finalizado ou não, se é fixo ou não, qual função deve ser usada para percorrê-lo, etc. Objetos grandes são alocados em sua própria página, que também possui o bloco com as informações acima citadas. Existem ainda objetos especiais, classificados pela máquina virtual como fixos, que não são tratados pelo *Garbage Collector*. Esses objetos só podem ser liberados explicitamente pela máquina virtual.

Existem dois tipos de listas de blocos livres disjuntas. O primeiro consiste num *array* onde cada elemento é uma lista de blocos livres de tamanho pré-determinado. Esses tamanhos são escolhidos numa faixa em que os objetos são considerados pequenos, indo de 16 bytes a aproximadamente 4 KBytes. Essa configuração se baseia na hipótese, que se confirma na prática, de que a alocação de objetos pequenos é bem mais frequente que a de objetos grandes [18]. Tornando direta a escolha de um bloco livre para a alocação de um objeto desse tipo estamos acelerando o processo para o caso mais frequente, consequentemente melhorando o desempenho com relação ao tempo gasto no processo de alocação. Esta é uma das maneiras tidas como mais eficientes para equiparar o desempenho de coletores que provocam a fragmentação do *heap* com o de coletores que realizam a compactação do mesmo. O segundo tipo é uma lista ligada de blocos de memória livre, sem tamanho pré-definido, usados para alocar objetos grandes e para reposição nas listas de blocos pequenos.

A implementação do coletor de lixo na JVM *Kaffe* é feita de tal forma que existe uma interface entre ele e a máquina virtual. Isto implica que existe um conjunto de funções que devem estar obrigatoriamente presentes, pois serão utilizadas para a interação com outras partes da *Kaffe*. Isto compõe uma *Application Program Interface (API)*, que vem proporcionar a liberdade de que, para trocar o sistema de coleta de memória, o programador tem que ter somente a preocupação de oferecer todas as funcionalidades exigidas nessa interface. Não existe exigência alguma sobre qual algoritmo ou a maneira como este deve ser implementado. Podemos ver as funções componentes dessa API na Tabela 2.1.

Pela Tabela 2.1, a função *gcWalkMemory* é chamada para percorrer um objeto. A máquina virtual *Kaffe* possui funções de percurso específicas para cada determinado tipo de objeto. A função *gcWalkMemory* retira a informação necessária do bloco existente no início da página para identificar qual a função apropriada para o seu percurso. Essa

Função	Descrição
gcMalloc	Chama alocador de memória
gcRealloc	Realoca objeto em nova área
gcFree	Libera um objeto fixo
gcInvokeGC	Invoca o Coletor de Lixo
gcInvokeFinalizer	Invoca o Finalisador
gcInit	Realoca objeto em nova área
gcEnable	Cria Thread do GC
gcMarkAddress	Marca um endereço de memória. O argumento é testado para saber se aponta para um objeto
gcMarkObject	Marca um objeto. Assume que o argumento aponta para um objeto
gcGetObjectSize	Retorna o tamanho do objeto
gcGetObjectDescription,	Retorna uma descrição de tipo do objeto. Usada para depuração.
gcGetObjectIndex	Retorna o índice da função para percorrer o objeto
gcWalkMemory	Percorre o objeto chamando a função apropria a seu tipo
gcWalkConservative	Percorre a memória conservativamente
gcRegisterFixedTypeByIndex	Registra os índices das funções de finalização, destruição e percurso de um tipo de objeto fixo
gcRegisterGcTypeByIndex	Registra os índices das funções de finalização, destruição e percurso de um tipo de objeto coletável

Tabela 2.1: API para o coletor da JVM *Kaffe*

informação é fornecida pela máquina virtual no momento da alocação do objeto.

Note que, apesar dessa liberdade, o esquema de alocação de memória adotado, com páginas de objetos de tamanho fixo e blocos de identificação para cada página, impede a implementação de um coletor que movimente os objetos ao longo do *heap*.

Capítulo 3

Experimentos com Codificação Implícita

Numa primeira etapa deste projeto, realizamos um estudo visando elaborar um esquema de codificação dos endereços de objetos vivos no *heap*. O objetivo dessa codificação foi a extração de informações úteis à compactação sem a necessidade de atravessarmos o grafo de objetos vivos.

A idéia foi utilizarmos o conceito de *Binary Decision Diagram (BDD)* [12] para codificar os endereços dos objetos vivos no *heap*. BDD é uma representação estrutural de funções booleanas, que vem sendo utilizada com sucesso na solução de problemas em desenvolvimento e análise de circuitos digitais, otimização matemática e inteligência artificial [13] [17]. A partir da representação em BDD do *heap*, podemos, por exemplo, saber a ordem dos objetos alocados e o tamanho do espaço entre eles. A compactação seria então realizada sem atravessar todo o *heap*, utilizando-se um apontador para a próxima posição livre ($free$) e os dados da BDD. Esse processo se daria como descrito no Algoritmo 3.

Algoritmo 3 Compactação Usando Codificação

function Compact()

$free = botton$
for Every address in BDD

Move its contents to $free$
$free += object\ size$

Uma BDD é uma forma de representação canônica de funções booleanas. As funções são representadas através de grafos dirigidos acíclicos, onde seus vértices representam

x1	*x2*	*x3*	*f*
0	0	0	0
0	1	0	0
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0
1	0	1	1
1	1	1	1

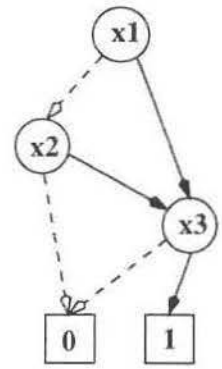


Figura 3.1: Exemplo de BDD

variáveis lógicas da função dada. A BDD é composta de nós intermediários e de dois nós terminais, $ZERO$ e UM. Cada nó intermediário corresponde a uma variável chamada de *variável topo*. Um caminho percorrido a partir da primeira variável na BDD levará ao nó $ZERO(UM)$ se e somente se a sua função lógica associada resultar em *falso(verdadeiro)* para uma dada atribuição às variáveis da mesma. A Figura 3.1 mostra uma função booleana dada pela sua tabela verdade e codificada através de uma BDD.

A codificação do *heap* através de uma BDD se dará da seguinte forma. Suponha, para simplificar, um espaço de endereçamento de 3 bits, e que os registros 101, 001 e 010 foram identificados como sendo vivos. A BDD que armazena essa informação seria dada pela Figura 3.2, onde b_1, b_2 e b_3 são as variáveis que representam os bits do endereço (do mais para o menos significativo). Os endereços de todos os três registros vivos podem ser obtidos atravessando-se a BDD a partir do vértice final UM até a raiz. Cada caminho fornece um endereço de registro vivo. Na BDD, o caminho tracejado (contínuo) indica o nó seguinte caso a variável sendo testada assuma o valor *ZERO(UM)*.

Agora suponha que encontramos mais um registro vivo, cujo endereço é 110. Para incluirmos esse endereço na BDD devemos fazer uma operação de OR entre a BDD que representa esse endereço como sendo o único vivo e a BDD que já contém todos os outros endereços vivos. Como resultado teríamos a BDD dada pela Figura 3.3, que neste caso, é menor que a BDD inicial.

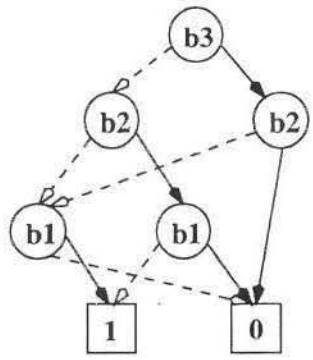


Figura 3.2: Registros vivos: 101, 001, 010

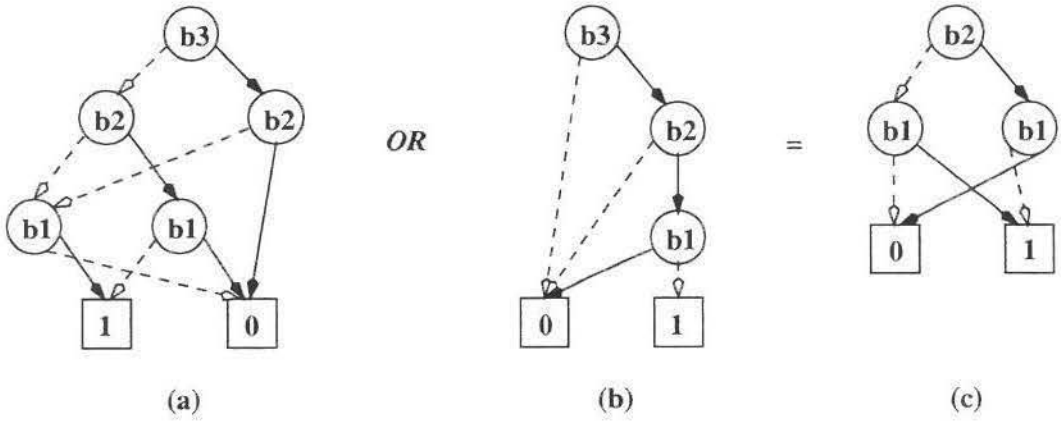


Figura 3.3: Resultado: Registros vivos 101, 001, 010 e 110

3.1 O Operador ITE

Para a implementação dos algoritmos de manipulação das BDDs utilizamos uma técnica introduzida por Brace, Bryant e Rudell em [11]. Eles apresentam um operador chamado *ITE (If Then Else)* para realizar as operações lógicas entre BDDs. Esse algoritmo permite a realização de todas as operações lógicas possíveis entre duas funções booleanas.

O operador ITE é uma função booleana definida para três entradas F, G, H e calcula *if F then G else H*. Isso é equivalente a:

$$ite = F.G + \overline{F}.H$$

Para fins computacionais, foi elaborada uma formulação recursiva para esse operador, baseada no *Teorema da Decomposição de Shanon*[11] :

$$F = v.F_v + \overline{v}.F_{\overline{v}}$$

onde $F_v = F$ quando $v = 1$ e $F_{\overline{v}} = F$ quando $v = 0$. F_v é conhecida como *cofator* de F para v.

Seja $F = (v, T, E)$ um nó da BDD, onde v é sua variável topo, T é seu valor caso $v = 1$ (filho direito) e E é seu valor caso $v = 0$ (filho esquerdo). A variável topo para um grupo de BDDs é dada pela menor variável topo entre elas. Seja $I = ite(F, G, H)$ e v a variável topo entre F, G, H, então:

$$\begin{aligned} I &= v.I_v + \overline{v}.I_{\overline{v}} \\ &= v.(F.G + \overline{F}.H)_v + \overline{v}.(F.G + \overline{F}.H)_{\overline{v}} \\ &= v.(F_v.G_v + \overline{F}_v.H_v) + \overline{v}(F_{\overline{v}}.G_{\overline{v}} + \overline{F}_{\overline{v}}.H_{\overline{v}}) \\ &= ite(v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \\ &= (v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \end{aligned}$$

Temos os seguintes casos terminais para essa recursão:

1. $ite(1, G, H) = G$
2. $ite(0, G, H) = H$
3. $ite(F, 1, 0) = F$

A principal razão para a utilidade das BDDs é a sua forma canônica. Para garantir que essa característica será respeitada na construção de BDDs através desse operador é mantida uma *hash-table*, chamada *unique_table*. Essa tabela associa cada tripla (v, G, H) a um nó $F = (v, G, H)$, sendo consultada cada vez que um nó está para ser criado afim

de evitar que existam dois nós idênticos ou seja, com descendentes idênticos, na BDD. Uma segunda *hash-table*, chamada *computed_table*, melhora o desempenho do algoritmo evitando que uma operação seja efetuada mais de uma vez. Ela mapeia três nós F, G, H ao nó resultado $ite(F, G, H)$. O Algoritmo 4 nos mostra como é implementado esse operador.

Podemos destacar alguns pontos importantes nesse algoritmo:

- Linha (2): Teste para verificar se a chamada não representa um dos caso terminais descritos anteriormente.
- Linha (4): Consulta a tabela *computed_table*, para verificar se o cálculo de $ite(F, G, H)$ já não foi realizado. Em caso de resposta positiva, retorna o resultado evitando uma computação dobrada.
- Linha (8): Chamada recursiva para o cálculo do filho direito (Then) do nó resultado da chamada atual.
- Linha (9): Chamada recursiva para o cálculo do filho esquerdo (Else) do nó resultado da chamada atual.
- Linha (10): Se um o mesmo nó será filho esquerdo e direito do resultado da operação atual, então ele será o resultado dessa operação. Mantém a forma canônica da BDD.
- Linha (11): Consulta a tabela *unique_table*, para verificar se o nó que está para ser criado como resultado dessa operação já não existe. Em caso de resposta positiva, retorna o resultado evitando a criação de nós dobrados. Mantém a forma canônica da BDD.

3.2 Codificando o *Heap* Implicitamente

Um experimento inicial foi realizado visando construir BDDs para configurações do *heap* de programas executados na JVM. O objetivo aqui era associar variáveis ($b_0, b_1, ..., b_{31}$) aos bits do endereço de cada objeto vivo, e codifica-los como descrito no início desse capítulo.

A BDD é construída após a fase de identificação dos objetos vivos, fase de marcação no caso do algoritmo *Mark-Sweep*. Para isso, implementamos o operador ITE, descrito na Seção 3.1, visando realizar as operações de OR entre BDDs.

Como era esperado, a *BDD* resultante que codifica os endereços de todos os objetos vivos se mostrou pequena. Isso devido ao número reduzido de variáveis, uma para cada bit do endereço, e à forma especial das *BDDs* que codificam apenas um endereço, onde

Algoritmo 4 Algoritmo para o Operador ITE

```
function ite(F,G,H)                                          (1)
    if( terminal case )                                      (2)
        return result;                                       (3)
    else if computed_table has entry (F,G,H)                 (4)
        return result;                                       (5)
    else                                                     (6)
        let v be the top variable of (F,G,H);                (7)
        T = ite(F_v, G_v, H_v);                              (8)
        E = ite(F_v̄, G_v̄, H_v̄);                              (9)
        if( T == E ) return result;                          (10)
        R = find_or_add_unique_table(v,T,E);                 (11)
        insert_computed_table((F,G,H),R);                    (12)
        return R;                                            (13)
```

cada nó possui um filho como sendo um nó intermediário (ou o terminal UM no caso do último bit) e o outro obrigatoriamente sendo o nó terminal ZERO. Isso acontece pois cada bit possui apenas um valor possível para cada endereço. Um exemplo de BDD nessa forma é a BDD codificando o endereço `110` na Figura 3.3(b). Chamaremos essas BDDs que representam apenas um endereço de objeto vivo como BDDs auxiliares no restante desse texto.

Entretanto, essa mesma forma particularizada fez com que o lixo gerado após cada operação de OR entre *BDDs*, como as da Figura 3.3, fosse muito grande. Isso porque os nós das BDDs auxiliares não eram reaproveitados para a construção da BDD resultado. Os dados contidos na Tabela 3.1, obtidos de coletas executadas em dois programas diferentes, ilustram a quantidade de lixo gerada na construção da BDD.

A memória ocupada por cada nó da BDD é de 20 bytes. A partir da Tabela 3.1, nota-se que a BDD resultado ocupa em torno de 10 Kbytes, mas considerando todo o processo de codificação foram alocados em torno de 400 Kbytes para o primeiro programa e de 560 Kbytes para o segundo. O *heap* de execução dessas aplicações possui 2 Mbytes, o que mostra que poderíamos guardar toda informação calculada ocupando 0,5% do *heap* para essas coletas, entretanto todo o processo utilizaria cerca de 25% dessa memória, o que o torna inviável. Em vista desses resultados começamos a alterar os algoritmos, tentando diminuir a quantidade de nós intermediários necessários.

Programa	# Objetos Vivos	# Nós na BDD Resultado	Total de Nós Criados
1	607	455	27310
1	630	474	28317
2	630	463	17024
2	679	500	18208

Tabela 3.1: Primeiros Resultados da Codificação

3.3 Operador ITE Especializado para OR

A implementação do operador ITE descrita anteriormente, realiza todas as operações lógicas possíveis entre duas funções booleanas. Para nosso projeto precisávamos exclusivamente da operação de OR entre BDDs, e ainda assim possuíamos a informação extra de que uma das BDDs sempre teria a forma especial descrita anteriormente.

O algoritmo do operador ITE foi especializado para explorar as características específicas de nossas BDDs. Em termos desse operador, a operação de OR entre duas funções booleanas, F e H, é dada por: $ite(F, G, H)$ fazendo-se $G = 1$, isto é, se F é verdadeira então a operação já é verdadeira, caso contrário o resultado da avaliação de H será o resultado da operação.

Como estamos interessados apenas em realizar operações de OR, uma especialização imediata para nosso algoritmo é o fato de G ser sempre verdadeiro (1). Então foram imediatamente desconsiderados os detalhes de implementação para o caso onde $G \neq 1$.

A segunda especialização implementada no algoritmo está relacionada à estrutura dos endereços de memória. Como estamos trabalhando apenas com uma faixa de endereços de memória, referente ao *heap*, pudemos constatar que todos os endereços possuíam muitos bits em comum, principalmente em se tratando dos bits mais significativos do endereço, por exemplo, dependendo da região de memória e do tamanho do *heap*, chegamos a verificar, muitas vezes, que os sete bits mais significativos de todos os endereços seriam sempre *zero*. Isto significa que, baseados na localização e tamanho do *heap*, podíamos diminuir ainda mais o número de variáveis de interesse em nosso problema.

Finalmente, como uma última especialização ao operador, passamos a não construir a BDD auxiliar, que representa apenas um endereço a ser incluído na BDD resultado, criando apenas os nós que efetivamente fariam parte da BDD resultado. Isso é possível porque, em nosso problema, os casos terminais da recursão são:

1. $ite(1, 1, H) = 1$
2. $ite(F, 1, 0) = F$

3. $ite(0, 1, H) = H$

onde H é a BDD auxilar que codifica o endereço a ser incluído na BDD resultado. Veja que os nós de H somente farão parte do resultado no terceiro caso. Por exemplo, supondo que esse caso terminal ocorra quando a recursão chegou ao nível do décimo bit do endereço, todos os nós que fossem criados na construção de H para os bits mais significativos que 10 teriam se tornado inúteis, gerando lixo. Portanto, passamos a construir somente os nós da BDD auxiliar a partir do nível onde ocorreu esse caso terminal.

Foi possível a execução do algoritmo sem os nós intermediários da BDD auxiliar devido à estrutura especial mencionada anteriormente. Essa estrutura, onde um dos filhos sempre aponta direto ao nó terminal $ZERO$, pode ser constatada nas BDDs das Figuras 3.3(b) e 3.4(b). Isso fez com que pudéssemos substituir o uso desses nós por consultas ao array onde estava armazenado o endereço. Para cada chamada recursiva do algoritmo em um nó intermediário, precisamos de duas informações que estariam armazenadas num nó da BDD auxiliar:

a) O valor de sua variável topo(v), que está armazenado na posição do array correspondente ao bit do endereço.

b) Os filhos esquerdo e direito dado que $v = 1$ e dado que $v = 0$ Esta informação é calculada da seguinte maneira:

- Se $v = 1$: $THEN(filho\ direito)$ é o próximo elemento do array, isto é, o "nó" que representa o próximo bit do endereço, e
  $ELSE(filho\ esquerdo)$ = Nó terminal $ZERO$.
- Se $v = 0$: $THEN$ é nó terminal $ZERO$ e $ELSE$ é o próximo elemento do array.

Desta maneira, podíamos calcular as informações necessárias ao algoritmo *sem* construir os nós da BDD auxiliar. Esses nós se tornaram necessários apenas quando a recursão alcançava o caso terminal 3.

Para esclarecer o funcionamento dessa nova versão do algoritmo, vamos acompanhar a execução de um exemplo. Suponha que temos dois objetos vivos e que estamos trabalhando com endereços de cinco bits. Os endereços são 10010 e 11001 e estão codificados pela BDD na Figura 3.4(a) da página 37. Queremos incluir mais um endereço de objeto vivo (10101) nessa BDD. Para isso, teríamos que construir a BDD dada na Figura 3.4(b) e realizar a operação de OR entre as BDDs (a) e (b) da referida Figura. Ao invés de construirmos a segunda BDD, vamos utilizar o operador ITE especializado, que doravante denominaremos apenas de *or*. Esse novo operador trabalhará com o array contendo os bits do endereço 10101, ilustrado na Figura 3.4(c), e com a BDD da Figura 3.4(a).

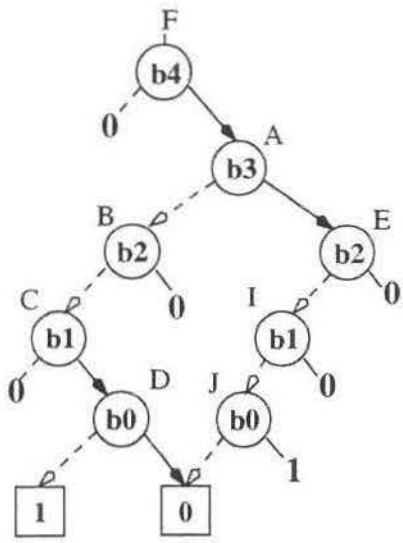
Convém descrevermos a notação usada nos algoritmos e no exemplo para facilitar a compreensão dos mesmos:

- BDD_ZERO: representa a BDD formada apenas pelo nó $ZERO$, que é um dos nós terminais da BDD resultado;
- BDD_ONE: representa a BDD formada apenas pelo nó UM, que é o outro nó terminal da BDD resultado;
- v: variável topo de um nó da BDD;
- F_v e $F_{\overline{v}}$: Função F calculada dado que $v = 1$ e $v = 0$ respectivamente;
- $(H)_v$ e $(H)_{\overline{v}}$: Função H calculada dado que $v = 1$ e $v = 0$ respectivamente, sendo que os parênteses indicam nós da BDD auxiliar não construídos. As informações são necessárias calculadas como descrito anteriormente.
- $or(F, H)$: Chamada à função que implementa o operador *ite* especializado para realizar a operação de OR entre as BDDs F e H, onde H representa a BDD auxiliar que, como descrito acima, não será explicitamente construída, com excessão aos nós que se mostrarem necessários ao final da operação.
- (v, T, E): Nó de uma BDD cuja variável topo é v, filho direito ($v = 1$) é T e filho esquerdo ($v = 0$) é E.

A função $or()$ (Algoritmo 5) recebe um apontador para um nó de BDD e um inteiro, que é a posição no *array* correspondente ao bit do endereço que deveria estar codificado numa BDD auxiliar H. Vale ressaltar aqui que uma chamada $OR(F, H)$ é equivalente a $ite(F, G, H)$, com $G = 1$. A BDD G foi descartada por ser constante na operação de OR, como explicado anteriormente.

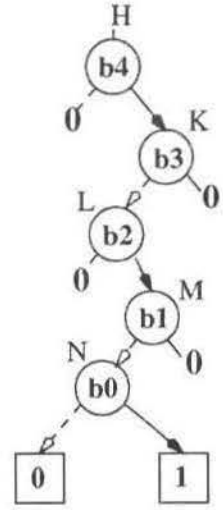
Representaremos os nós da BDD da Figura 3.4(b) entre parênteses (como (X)). Acompanhando as chamadas recursivas de $or()$ vemos que:

$$
\begin{aligned}
or(F,H) &= or = (b_4, or(F_{b_4}, (H)_{b_4}), or(F_{\overline{b_4}}, (H)_{\overline{b_4}})) && (1)\\
&= (b_4, \underbrace{or(A,(K))}_{O}, \underbrace{or(0,0)}_{caso\ terminal\ 2}) && (2)\\
O = or(A,(K)) &= (b_3, or(A_{b_3}, (K)_{b_3}), or(A_{\overline{b_3}}, (K)_{\overline{b_3}})) && (3)\\
&= (b_3, \underbrace{or(E,0)}_{caso\ terminal\ 2}, \underbrace{or(B,(L))}_{P}) && (4)\\
P = or(B,(L)) &= (b_2, or(B_{b_2}, (L)_{b_2}), or(B_{\overline{b_2}}, (L)_{\overline{b_2}})) && (5)
\end{aligned}
$$



Endereços Vivos: 10010 e 11001

(a)

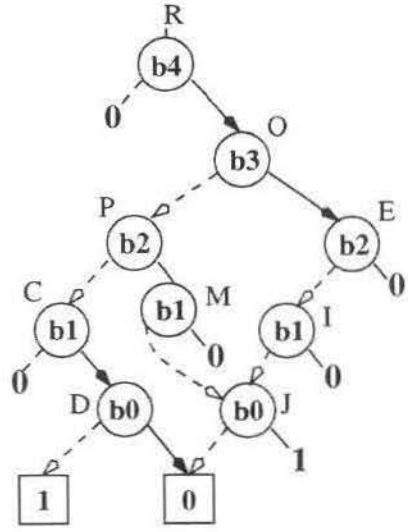


Endereço Vivo: 10101

(b)

H	K	L	M	N
1	0	1	0	1

(c)



Endereços Vivos: 10010, 11001 e 10101

(d)

Figura 3.4: Endereços Codificados

$$= (b_2, \underbrace{or(0,(M))}_{caso\ terminal\ 3}, \underbrace{or(C,0)}_{caso\ terminal\ 2}) \qquad (6)$$

Então: $R = or(F,H) = (4, O, ZERO)$

A Figura 3.4(c) mostra o array contendo o endereço a ser incluído na BDD. Nessa figura, as letras sobre os elementos indicam qual nó da BDD da Figura 3.4(b) esses elementos estão representando.

Ilustraremos a utilização dos elementos do array ao invés da construção direta da BDD da Figura 3.4(b), utilizando a linha (3) do esquema de recursão apresentado acima. Nessa linha, precisávamos calcular $or(A_{b_3}, (K)_{b_3})$ e $or(A_{\overline{b_3}}, (K)_{\overline{b_3}})$. Pela BDD da Figura 3.4(a) obtemos:

$$A_{b_3} = E \text{ e } A_{\overline{b_3}} = B$$

Agora, obtemos os filhos do nó (K) pelo array. Observe que o valor armazenado no `segundo elemento` do array, que representa o bit número 3 do endereço, é `0`. Isso implica, visto que cada bit só pode ter um valor válido, que o filho direito de (K) é o nó terminal $ZERO$, e portanto, o próximo elemento do array representa o nó (L), filho esquerdo de (K). Isso nos deixa com:

$$(K)_{b_3} = 0 \text{ e } K_{\overline{b_3}} = L \text{ e, portanto:}$$
$$or(A_{b_3}, (K)_{b_3}) = or(E, 0) \text{ e } or(A_{\overline{b_3}}, (K)_{\overline{b_3}}) = or(B, (L))$$

Note que, a única informação necessária para decidir se o próximo elemento do array é o filho esquerdo ou direito do elemento atual foi o valor de sua variável, os nomes dos nós (K) e (L) foram usados para maior clareza, possibilitando ao leitor visualizar em que ponto estamos na BDD da Figura 3.4(b). Essa é justamente a única informação que precisamos para avançar mais um nível na recursão.

A Figura 3.4(d) mostra a BDD R, resultado da operação $or(F,H)$, cujo nó inicial é $(b_4, O, ZERO)$. Note que só foram necessários nós da BDD auxiliar no caso terminal alcançado na recursão que calcula o nó `P`. Somente o nó `M` da BDD H foi construído (correspondente á variável b_1 do endereço). O nó N não precisou ser construído pois é idêntico ao nó J que, portanto, foi reaproveitado para manter a forma canônica da BDD. Nos casos reais, com endereços de 32 bits, foi frequente a necessidade de construção da BDD auxiliar apenas a partir do vigésimo bit ou valores ainda menores. Depois de algumas operações de OR consecutivas, cresce o reaproveitamento de nós atráves da *unique-table*, evitando assim a construção de mais alguns nós. No nosso algoritmo para o operador *or*, descartamos a outra *hash-table*, chamada *computed-table*, que visa evitar a computação repetidamente de um mesmo nó. Observando a execução, notamos que isso quase não

ocorria no nosso caso, fazendo com que não se justificasse o custo de manutenção da tabela.

Ainda assim, alguma quantidade de lixo continua sendo gerada. No nosso exemplo, os nós A, B e F da BDD dada na Figura 3.4(a) não foram aproveitados na BDD resultado R. Isso porque fizeram-se necessários nós representando esses mesmos bits, cujos filhos fossem diferentes dos já existentes.

Algoritmo 5 Operador OR (ITE Especializado)

```
function or( node* F, int H)

    if(is_terminal(F, H)) return;
    if( Bit_value == 1)

        T = ite(F_{v=1}, H - 1);
        E = ite(F_{v=0}, BDD_ZERO);

    else

        T = ite(F_{v=1}, BDD_ZERO);
        E = ite(F_{v=0}, H - 1);

    if( T=E ) return;
    R = find_or_add_unique_table(v, T, E);
    return R;
```

Algoritmo 6 Teste de casos terminais

```
function is_terminal( node* F, int H)

    /* ite(1,1,H) */
    if(F == BDD_ONE)) return BDD_ONE;
    /* ite(F,1,0) */
    if(H == BDD_ZERO)) return F;
    /* ite(0,1,H) */
    if(F == BDD_ZERO))

        return BuildBDD(H); /* Build useful part of H BDD */

    return NULL;
```

Variável (Bit)	Número de Nós Alocados	
	Sem Otimização	Com Otimização
3	2	2
4	4	4
5	10	10
6	42	42
7	232	217
8	468	435
9	657	542
10	814	556
11	967	567
12	1103	567
13	1178	567
14	1210	567
15	1211	567
16	1212	567

Tabela 3.2: Resultados da Otimização

3.4 Resultados Finais

Mesmo com a implementação dessas otimizações, o lixo não pôde ser reduzido a uma quantidade que tornasse viável a construção da *BDD* que representaria todos os objetos vivos sem um prejuízo considerável no desempenho da aplicação e um consumo exagerado de memória. A Tabela 3.2 mostra os resultados obtidos com as otimizações, em números de nós criados para cada variável numa determinada coleta e a Tabela 3.3 nos dá informações sobre essas coletas.

Podemos observar que a implementação das otimizações reduziu praticamente pela

Característica	Sem Otimização	Com Otimização
# Objetos Vivos	607	567
# Nós na BDD Resultado	455	428
Total de Nós Criados	27310	13720
Total de Memória	533,4 Kb	267,9 Kb
Memória usada na BDD Resultado	8,8 Kb	8,35Kb

Tabela 3.3: Características das Coletas Utilizadas

metade a quantidade de lixo criada durante o processo, mas ainda assim a quantidade de memória total utilizada permaneceu grande comparada com a necessária para armazenar o resultado da codificação. O lixo remanescente é proveniente de nós da BDD resultado que se tornaram inúteis, sendo que esse tipo de lixo nós não temos como prevenir por ser dependente dos resultados das operações de *OR*.

O único jeito de diminuir ainda mais a quantidade de memória desperdiçada seria implementar um esquema para reaproveitar os nós que se tornam lixo. Seria praticamente um esquema de *Garbage Collection* exclusivo para a BDD. Apesar de ser um processo bem mais simples do que um *Garbage Collector* normal, o prejuízo causado no desempenho da aplicação o tornaria inviável.

Esses fatores nos fizeram decidir pela mudança na abordagem escolhida e basear o nosso algoritmo de compactação em informações retiradas do próprio *heap*.

Capítulo 4

Experimentos com Compactação

O principal objetivo da segunda fase de nosso projeto foi a implementação de um sistema automático de gerenciamento de memória, para a máquina virtual Java, que fosse capaz de realizar a compactação dos dados no *heap*.

Iniciamos um estudo dos métodos de compactação de memória existentes na literatura e das características de comunicação entre a máquina virtual *Kaffe* e seu coletor, afim de determinar um algoritmo de compactação que se adaptasse à máquina virtual em questão.

À princípio, não sabíamos qual seria o impacto da introdução de um esquema para compactação de memória no desempenho do coletor. Uma maneira que escolhemos para evitar uma queda considerável no desempenho foi implementar um sistema de gerenciamento de memória que possibilitasse uma abordagem híbrida ao problema de coleta de lixo. Essa abordagem possui a seguinte característica: implementaríamos um coletor capaz de realizar coletas baseadas no tradicional algoritmo *Mark-Sweep* bem como coletas baseadas no algoritmo *Mark-Compact*.

Isso veio nos proporcionar uma maior liberdade na configuração de nosso coletor: ele é capaz de intercalar coletas com compactação dos dados entre coletas tradicionais, sendo que a frequência com que a compactação é realizada pode ser calibrada de forma a não causar grande impacto no desempenho e não permitir um acúmulo de fragmentação.

4.1 Projeto do Alocador de Memória

Para acabar com a restrição de posicionamento dos objetos, o novo alocador armazena os dados relativos a cada objeto em cabeçalhos que ficam localizados no endereço de memória imediatamente anterior ao objeto em si. Também não existe mais a divisão do *heap* em páginas, possibilitando assim a qualquer objeto ocupar qualquer posição no *heap*. Sendo assim, cabeçalho e objeto passam a ser um único bloco a ser movimentado, quando necessário, pelo coletor. O conteúdo desse cabeçalho está descrito na Tabela 4.1.

Campo	Descrição	Tamanho
next	Apontador para o próximo elemento em uma lista	4
prev	Apontador para o elemento anterior em uma lista	4
size	Tamanho do objeto	4
func	Índice da função usada para percorrer o objeto	2
state	Indica a cor do objeto e se é finalisável ou não	2
inuse	Indica que esse bloco está em uso	4
forward	Endereço para onde será movido o objeto	4
TOTAL		24

Tabela 4.1: Conteúdo do Cabeçalho dos Objetos

O sistema que implementamos em nosso alocador utiliza um esquema conhecido como alocação de dois níveis [22]. Esse esquema trabalha com duas listas de blocos livres: uma sendo um vetor de listas de blocos com tamanhos pré-determinados, usada para alocação de objetos pequenos (a qual chamamos de lista de primeiro nível) e a outra sendo uma lista ligada de blocos sem tamanhos pré-definidos que são utilizados para alocação de objetos grandes e para a realimentação das listas do primeiro nível (chamada de lista de segundo nível). Isto tudo visando agilizar a alocação dos objetos pequenos, que são mais frequentes, evitando que seja necessária uma busca através dos blocos livres para encontrar um de tamanho adequado. Essa configuração é similar à usada pelo alocador original da máquina virtual *Kaffe*, a menos da paginação do *heap*.

Note, pelo Algoritmo 4.1, que nosso alocador já está preparado para tirar proveito dos benefícios trazidos pela compactação de memória. O primeiro passo do processo de alocação é verificar se existe espaço livre suficiente no topo do *heap* para acomodar o objeto. Se a resposta for positiva a alocação será simplesmente um incremento, do tamanho do objeto, no apontador que indica o topo do *heap*. Caso a resposta seja negativa entramos no processo de alocação que trabalha com listas de blocos livres em dois níveis. Como descrito acima, o algoritmo tenta alocar os objetos pequenos na lista correspondente ao seu tamanho, caso não haja mais blocos livres nessa lista ela é realimentada com blocos vindos da lista de segundo nível. Em se tratando de um objeto grande, a alocação se dá diretamente a partir da lista de segundo nível. Pelo Algoritmo 8, vemos que se esta lista não possuir um bloco de tamanho suficiente, o alocador tenta obter mais memória invocando o coletor de lixo como uma primeira tentativa. Caso a coleta não recupere uma quantidade suficiente de memória, o alocador é forçado a requisitar mais memória ao sistema operacional.

Com isso, nosso sistema de alocação está capacitado a trabalhar com um *heap* frag-

mentado ou compactado, concordando com nossa proposta de elaborar um sistema híbrido onde não é assegurado que o *heap* estará compactado o tempo todo.

A máquina virtual *Kaffe* aloca objetos que não são tocados pelo coletor. Ela classifica esses objetos como não tratáveis pelo coletor, ou seja, somente podem ser liberados explicitamente pela máquina virtual. No sistema de gerenciamento de memória original esses objetos estão armazenados ao longo do *heap* de dados. Isso consistiria num problema para a compactação, pois esse tipo de objeto nos impede a movimentação dos outros objetos ao longo do *heap*, impossibilitando a compactação de memória. Para solucionar esse problema, implementamos o *heap* dividido em duas partes: uma contendo somente os objetos que não poderiam ser utilizados pelo coletor, e outra contendo os objetos que poderiam ser liberados e movidos pelo sistema de coleta de lixo.

A alocação de memória nessa região separada do *heap* se dará da mesma maneira descrita acima. Ela possui sua própria lista de blocos livres em dois níveis, com a diferença que essa lista só é realimentada quando a máquina virtual libera explicitamente um bloco, visto que não existe coleta para essa região.

Algoritmo 7 Alocação de Memória

```
function GC_Allocator( size )

    for( ; ; )

        if There is enough space at the heap top
            p = top
            top = top + size
            return p
        else
            if It is a small Object
                Get first level freelist's index for this size

                if There is a free block in this list or in a
                larger one
                  p = &block
                  return p
                else
                  // Request a block from the second level freelist
                  p = Second_level_allocator( size );
            else
                p = Second_level_allocator( size );
```

Algoritmo 8 Alocação no Segundo Nível

function Second_level_allocator(*size* **)**

if There is a block that matches size

$p = \&block$
return p

else

if It is the first try on this level

call Garbage Collector
return

else

Request memory to Operating System

4.1.1 Alinhamento dos Objetos

Como explicado anteriormente, em nosso sistema de alocação, cada objeto está imediatamente precedido por seu cabeçalho na memória. Esse cabeçalho possui 7 campos e ocupa 24 bytes, como mostrado na Tabela 4.1.

A máquina virtual *Kaffe*, assim como a JDK1.1.5 VM, utiliza um alinhamento de 8 bytes para os objetos, isto é, todos esses objetos se iniciam em endereços múltiplos de 8, a partir da base do *heap*.

Do ponto de vista do sistema de recuperação de memória, esse alinhamento é muito importante no momento de se percorrer um objeto conservadoramente. Quando um apontador é passado para a função *gc_WalkConservative*, vários testes são realizados para se constatar quando esse apontador *pode* estar referenciando um objeto no *heap* ou não, e uma das principais exigências é a de que ele deve estar corretamente alinhado, dentro de sua página.

No nosso sistema de alocação, o *heap* não está dividido em páginas e cada objeto guarda consigo o cabeçalho, que possui tamanho múltiplo de 8 bytes, no caso, 24 bytes. Para podermos implementar a função conservativa para o percurso de um objeto, precisamos garantir que todos os cabeçalhos estão alinhados segundo algum critério que, para evitar causar alguma incompatibilidade com outras partes da máquina virtual, deve ser um múltiplo de 8 bytes. Isso nos levou a adotar o seguinte critério: todos os objetos deveriam ter um tamanho múltioplo de 24 bytes, isto é, a menor unidade de alocação, em nosso sistema, seria equivalente ao tamanho de um cabeçalho.

O teste conservativo, baseado no alinhamento dito acima, consiste basicamente da

verificação dos seguintes aspectos:

- Se o apontador referencia ou não um endereço múltiplo de 24, a partir da base do *heap*;
- Se o conteúdo desse apontador pode ser um cabeçalho de um objeto no *heap*, isto é, se o conteúdo de seus campos possuem valores válidos.

4.2 Projeto do Coletor

A primeira versão de nosso coletor manteve as características do original, inclusive a de ser baseado no algoritmo *Mark-Sweep*. A principal diferença é que foi adaptado ao novo alocador, isto é, passou a trabalhar com nosso novo esquema de cabeçalhos. Todas as informações necessárias à máquina virtual e a ele mesmo estão em campos desse cabeçalho, como consta na Tabela 4.1.

O sistema de gerenciamento de memória atual já está preparado para movimentar os objetos no *heap*. A partir desse ponto podemos implementar o algoritmo de compactação de memória.

4.2.1 Técnicas Para Compactação de Memória

Visando tratar o problema de fragmentação presente no algoritmo *Mark-Sweep*, foi implementado um esquema de compactação de memória para o coletor da JVM *Kaffe*. Esse esquema foi baseado no algoritmo *Mark-Compact*, inicialmente descrito na Seção 2.1.4.

Algoritmos de compactação executam vários passos sobre o *heap*. O número de passos varia de acordo com o algoritmo e da possibilidade de se realizar otimizações para combiná-los. Em geral, esses algoritmos possuem três passos:

1. Marcar a estrutura de objetos vivos;
2. Compactar os dados;
3. Atualizar os valores dos ponteiros para blocos que foram movidos.

A ordem de execução dos dois últimos passos varia de acordo com o algoritmo. Vale ressaltar que não estamos considerando nessa análise algoritmos que realizam a compactação através da cópia de objetos entre semi-espaços.

A ordem resultante dos objetos, após feita a compactação, é um fator importante no desempenho da memória virtual e no número de *cache hits*. Com relação a esse fator, os algoritmos de compactação podem ser divididos em três classes:

- **Arbitrary:** Os blocos são movidos sem levar em consideração a sua ordem original ou o fato de possuírem apontadores entre si. Tais métodos podem ser mais simples de implementar e mais rápidos de serem executados, principalmente se todos os objetos forem de um tamanho fixo, mas geralmente resultam numa perda de localidade espacial.

- **Linearing:** Blocos que originalmente apontam entre si são movidos para posições adjacentes de memória. Um exemplo dessa classe são os algoritmos baseados em *Copy Collection* que atravessam o *heap* usando busca em profundidade.

- **Sliding:** Blocos são movidos para um extremo do *heap*, sobrepondo-se aos blocos livres (deixados pelos objetos mortos), entretanto mantendo sua ordem de alocação.

Os dois últimos estilos de compactação são muito mais eficientes que o primeiro em relação à memória virtual e à *cache*, pois visam melhorar a localidade de referência.

Existem outros fatores a serem considerados, no caso de compactação de objetos de tamanhos diferentes: quantos passos são necessários para mover os objetos e atualizar os apontadores; quanto de espaço extra é necessário; se existe algum tipo de restrição aos apontadores (Ex.:É permitido apontar para objetos já movidos?); e quanto de trabalho é realizado em cada etapa do algoritmo.

Podemos encontrar na literatura muitos algoritmos e otimizações para realizar compactação do *heap*. Em [22] podemos encontrar a seguinte classificação, que cobre uma quantidade representativa desses algoritmos.

- **Two-Finger Algorithms:** Utilizam-se de dois ponteiros, um para a próxima posição livre do *heap* e outro para o próximo objeto a ser movido. Para cada objeto movido um endereço de *forward* é deixado em seu antigo endereço. Tais métodos geralmente só podem ser aplicados para compactação de objetos de tamanho fixo.

- **Forwarding Address Algorithms:** É necessário um campo adicional no cabeçalho dos objetos para armazenar um endereço de *forward*. Esse campo guarda a posição de memória para onde vai ser compactado o objeto antes do mesmo ser movido. Estes métodos são aplicáveis a objetos de tamanho variado.

- **Table-based methods:** Uma tabela de transferência é contruída e consultada posteriormente para calcular os novos valores dos apontadores.

- **Threaded methods:** Cada objeto é ligado a uma lista de objetos que originalmente apontavam para ele. Quando o objeto é movido, essa lista é percorrida e os valores dos apontadores são atualizados.

4.2.2 Algoritmo Lisp 2

O algoritmo de compactação implementado para nosso coletor foi baseado num algoritmo apresentado em [22], chamado de *Lisp 2* (ver Algoritmo 9). Esse algoritmo é classificado como *Sliding*, visto que preserva a ordem de alocação dos objetos, e como *Forwarding address* por armazenar o endereço de destino dos objetos em seus cabeçalhos e trabalhar com blocos de tamanho variado. A fase de compactação realiza três passos no *heap* e exige um campo para armazenar o endereço de destino do objeto, que normalmente também é utilizado como campo de marcação. Nesse método uma coleta fica dividida em 4 fases:

1. **Mark:** Essa primeira etapa é idêntica à presente em coletas seguindo o algoritmo de *Mark-Sweep* tradicional e consiste na identificação dos objetos vivos no *heap*.

2. **Compute Address:** Nessa etapa são calculados os novos endereços dos objetos vivos, armazenado-os no campo *forwarding_address* existente no cabeçalho dos objetos (ver Algoritmo 11). Observe que o novo endereço é a soma, a partir da base do *heap*, dos tamanhos de todos os objetos vivos encontrados até agora. Esse endereço é apontado pela variável *free*. A função *Combine*() (Algoritmo 10) pode ser utilizada para juntar objetos mortos consecutivos em um único bloco para agilizar a travessia do *heap* nos passos seguintes.

3. **Update Pointers:** A terceira etapa atualiza os apontadores contidos nos objetos do *heap*(ver Algoritmo 12). Os apontadores são simplesmente trocados pelos novos endereços contidos no campo *forwarding_address* do cabeçalho dos objetos para os quais estão apontando. Note que não só apontadores entre objetos do *heap*, mas também as suas raízes devem ser atualizadas.

4. **Relocate Objects:** Na quarta etapa os objetos são movidos para seus novos endereços e o campo *forwarding_address* é zerado, preparando-o para a próxima coleta. Note, pelo Algoritmo 13, que a variável *free* apontará para a primeira posição livre do *heap* e, como todos os dados estão compactados, a alocação será simplificada como nos algoritmos baseados em *Copy Collection*.

4.2.3 Compactador JVMPack

Um coletor baseado nos algoritmos *Mark-Compact* ou *Copy Collection* tradicionais requer um conjunto de raízes bem conhecido. Isto significa que, em qualquer ponto da execução, o coletor deve ser capaz de identificar quais registradores ou em quais lugares da pilha de execução estão armazenados todos os apontadores para objetos no *heap*. Isso é feito, geralmente, extraindo-se o estado atual do programa de seus registradores e da pilha,

Algoritmo 9 Algoritmo de Compactação Lisp 2

```
function Compact_Lisp2()

    Mark();
    ComputeAddress;
    UpdatePointers();
    Relocate();
```

Algoritmo 10 União de Objetos Mortos

```
function Combine( P )

    next = P + size(P);
    while forwarding_address( next ) == NULL

        size(P) = size(P) + size(next);
        next = P + size(P);
```

Algoritmo 11 Calcula Novos Endereços para Objetos no Heap

```
function ComputeAddress()

    free = Heap_Bottom;
    P = Heap_Bottom;
    while P ≤ Heap_Top;

        if forwarding_address(P) ≠ NULL    /* Marked Object */
            forwarding_address(P) = free;
            free = free + size(P);
        else Combine(P)          /* Optional */
        P = P + size(P);
```

Algoritmo 12 Atualização dos Apontadores

```
function UpdatePointers()

    for R in roots

        R = forwarding_address(R);

    P = Heap_Bottom;
    while P ≤ Heap_Top;

        if forwarding_address(P) ≠ NULL

            for Q in Children(P)
            Heap[Q] = forwarding_address(Heap[Q]);

        P = P + size(P);
```

Algoritmo 13 Compacta Objetos no Heap

```
function Relocate()

    P = Heap_Bottom;
    while P ≤ Heap_Top;

        temp = P + size(P);
        if forwarding_address(P) ≠ NULL

            free = forwarding_address(P);
            forwarding_address(P) = NULL;
            move(P, free);

        P = temp;

    free = free + size(free);
```

examinando-se os *stack-frames*, variáveis locais e argumentos de funções e procedimentos. Essa exigência se faz necessária porque cada uma dessas raízes precisa ser atualizada, como consta no Algoritmo 12, com o novo endereço do objeto que foi movido. Coletores com essas características são conhecidos como coletores exatos. As pricipais maneiras do compilador fornecer essas informações são:

- **Tagging:** Torna os valores auto-descritivos, reservando um ou mais bits para indicar se o valor representa ou não uma referência. Requer a emissão de instruções extras para a manutenção dessas marcas.
- **Estruturas de Dados:** Pode-se utilizar estruturas de dados para descrição dos objetos. Nesse caso cada objeto começa com um apontador para um descritor do tipo do objeto, que deve conter um mapa de seus campos, possibilitando a identificação de seus apontadores. Se faz necessário também uma estrutura para a identificação dos apontadores na pilha de execução.

Porém, existem sistemas onde o coletor não tem como extrair todas as informações necessárias sobre as raízes. Nesses casos, é necessária a adoção de uma estratégia mais conservativa para esse conjunto de apontadores. Ao ínves de se exigir que todos os apontadores a partir das raízes para objetos no *heap*, e somente eles, estejam presentes num dado conjunto R, passa-se a exigir que R contenha somente *dicas* suficientes para se encontrar todos os dados acessíveis do *heap*. Dessa forma, todo objeto que *pode* estar sendo apontado por uma raiz será mantido pelo coletor, e não será movido para não causar qualquer inconsistência na pilha de execução. Em outras palavras, o coletor sabe que numa região de memória podem existir apontadores para objetos no *heap*, mas não sabe diferenciar entre inteiros e possíveis apontadores dentro dessa região. Um exemplo de coletor implementado utilizando-se esse tipo de abordagem é dado em [8], onde Bartlett implementa um algoritmo baseado em *Copy Collection* para coletar e compactar os dados num sistema onde as raízes não sejam bem conhecidas.

Um problema semelhante é encontrado na máquina virtual *Kaffe*. Nela existe um conjunto de apontadores que nos dá as raízes do programa, mas não temos como saber, por exemplo, a que células da pilha de execução correspondem esses apontadores. Isso significa que sabemos quais objetos do *heap* são apontados por variáveis da pilha de execução da máquina virtual, mas não temos como saber quais as variáveis da pilha que devem ser atualizadas, pois na *Kaffe* não existe nenhum tipo de diferenciação entre ponteiros e inteiros na pilha.

Existe ainda mais um fator: uma aplicação Java distribuída executa várias *threads*. De fato, o coletor da máquina virtual por si só é executado numa *thread* exclusiva. Isso implica que os apontadores que por acaso vierem a existir nas pilhas de execução

dessas *threads* também serão raízes para o coletor. Esse tipo de apontador é percorrido conservadoramente pelo coletor original da *Kaffe*, isto é, não existe maneira de determinar com precisão a diferença entre apontadores e números inteiros nessas pilhas de execução, então o coletor assume como sendo um apontador tudo que *puder* sê-lo.

Essas restrições com relação às raízes do *heap* nos obrigaram a adotar um esquema conservativo para a realização da compactação. Todos os objetos que estão sendo apontados pelas raízes, mais os objetos que *podem* estar sendo apontados pelas pilhas de execução das *threads*, serão considerados fixos no momento de uma compactação. A existência de objetos fixos no *heap* acaba criando alguns buracos mesmo após a compactação.

Como dito anteriormente, nosso alocador está apto a trabalhar tanto com o *heap* compactado como com o *heap* fragmentado, devido à nossa escolha de intercalar coletas de *Mark-Sweep* tradicional com coletas *Mark-Compact*. Isso nos possibilitou a elaboração de um algoritmo de compactação que identificasse os buracos deixados por esses objetos fixos conforme eles surgissem ao longo do processo e, ao término da compactação, acoplasse-os à lista de blocos livres para futuramente serem alocados. Portanto, após uma coleta, os objetos serão alocados no topo do *heap*, como num algoritmo baseado em *Copy Collection* e, a partir do momento que se esgote a memória disponível dessa maneira, os buracos deixados durante a compactação são aproveitados, reduzindo ao máximo a quantidade de memória desperdiçada devido à presença de objetos considerados fixos. Vale ressaltar aqui, que os objetos alocados pela máquina virtual que não são tratáveis pelo coletor, que também deveriam ser considerados como fixos, estão sendo alocados em uma área separada do *heap* contendo os objetos comuns, evitando assim a criação de novos buracos.

Esses fatores nos impediram de implementar a compactação utilizando os algoritmos na sua forma mais simples, como descritos na Seção 4.2.1. Tivemos que introduzir nos algoritmos já apresentados o tratamento especial dado aos objetos considerados fixos.

No momento da compactação, os objetos que serão considerados como fixos já estão identificados. Essa identificação ocorre na fase de marcação, enquanto o coletor está percorrendo as raízes do *heap* e toda vez que ele percorre algum apontador conservativamente. Foram introduzidas instruções nas funções da máquina virtual que percorrem esses tipos de objetos, para que armazenassem um código de identificação no campo $forward$ do cabeçalho dos mesmos à medida que os fossem percorrendo. Então, no momento de realizar-se o cálculo do novo endereço do objeto, o conteúdo desse campo é testado e, se for igual a esse código de identificação, o objeto permanece imóvel.

Vamos destacar agora as principais diferenças entre as funções do algortimo Lisp 2 e as implementadas em nosso coletor. As funções utilizadas em nosso compactador estão descritas pelos Algoritmos de 14 à 16.

- **Compute Address:** Nessa etapa são calculados os futuros endereços dos objetos vivos. Os objetos serão compactados no extremo inferior do *heap* até que surja um

objeto fixo. A partir daí, o buraco existente entre o atual topo do *heap* e o endereço do objeto fixo é armazenado numa lista de buracos e os novos endereços serão computados de uma maneira que, se algum objeto couber no buraco deixado, este será preenchido. Pelo Algoritmo 14, vemos que para cada objeto vivo encontrado no *heap*, o processo de calcular o seu novo endereço ocorre da seguinte forma:

1. O apontador `free` indica qual a próxima posição livre no *heap*, `h` aponta para o objeto sendo visitado e `top` aponta para o topo do *heap*, isto é, a última posição ocupada até o momento. Note que `free = top` enquanto não surgir um objeto fixo.
2. Se o objeto for fixo, então a variável `top` passa a apontar para a posição imediatamente posterior ao fim do objeto e nenhum endereço precisa ser calculado. Caso contrário, o novo endereço será calculado e armazenado no campo *forward* de seu cabeçalho.
3. A primeira opção é colocar o objeto na posição apontada por `free`. Note que, se `free` está apontando para o início de um buraco, pode ser que ele não caiba nessa posição. Se isso ocorrer `free` é movido para o início do próximo buraco, até que se encontre um em que o objeto se encaixa ou o topo do *heap*(*top*) seja atingido.
4. No caso do objeto não se encaixar em nenhum buraco entre `free` e `top` ele será movido para o endereço apontado por `top` e essa variável será incrementada de seu tamanho.

Os buracos que não são preenchidos nesse processo continuam armazenados numa lista para que, ao final da compactação, sejam encaminhados à lista de blocos livres correspondente ao seu tamanho. O processo se encerra quando todo o *heap* tiver sido atravessado em busca de objetos vivos a serem compactados.

- **Update Pointers:** Essa etapa de atualização dos apontadores contidos nos objetos do *heap*, não sofreu grandes alterações em comparação com o Algoritmo 12. Para nosso coletor não se fez necessária uma nova passagem por todo o *heap*. Isso se deve ao fato do coletor utilizar o esquema de marcação por três cores, descrito anteriormente, sendo que nesse ponto da coleta todos os objetos vivos se encontram na lista de pretos. Portanto podemos atualizar os apontadores necessários atravessando os objetos contidos nessa lista. No Algoritmo 15, pelos motivos já citados, também não aparece a atualização das raízes. Um ponto importante nessa etapa é a maneira de identificar os apontadores a serem ajustados. Como descrito na Seção 2.3, a máquina virtual possue funções apropriadas para percorrer cada tipo de objeto.

Essas funções identificam os apontadores que partem de um objeto e, dessa maneira, marcam seus descendentes. Utilizamos funções similares a essas para atualizar os apontadores onde, ao invés de marcar os descendentes, verificamos se ele vai ser movido. Se a resposta for afirmativa o apontador é alterado e passa a conter o novo endereço do objeto alvo. Note que em nosso algoritmo os apontadores são atualizados *antes* da movimentação dos objetos, isso porque o endereço de *forward* está armazenado no próprio objeto ao invés de ser deixado no antigo endereço, como em coletores baseados em *Copy-Collection*.

- **Move Objects:** Nessa fase os objetos são movidos para seus novos endereços e a lista de objetos pretos, que é uma lista duplamente ligada, tem seus apontadores atualizados. Para finalizar, os buracos que foram produzidos devido à presença dos objetos fixos são adicionados à lista de blocos livres (ver Algoritmo 16).

A Figura 4.1 ilusta um processo de compactação realizado em nosso sistema através dos algortimos descritos acima. Na Figura 4.1(a), temos o *heap* em seu estado inicial, logo após a fase de marcação. Podemos identificar dois objetos fixos(2 e 4) e três objetos vivos passíveis de compactação (1, 3 e 5).

Na Figura 4.1(b), o apontador h alcançou o objeto 1. Os apontadores $free$ e top referenciam a mesma posição: o início do *heap*. Então o novo endereço de 1 será igual a $free = top$. Esta é a situação descrita no item 1 acima.

No passo seguinte, visto na Figura 4.1(c), o apontador h alcançou o objeto 2. Note que os apontadores $free$ e top já estão atualizados, considerando o espaço que vai ser ocupado pelo objeto 1. Como 2 é um objeto fixo, nenhum endereço de forward é calculado. Esta é a situação descrita no item 2 acima.

Na Figura 4.1(d), o endereço de forward do objeto 3 é calculado. Note que o apontador top sempre aponta para a posição imediatamente acima do objeto já visitado que possui o maior endereço, e agora não coincide mais com $free$. Portanto, temos espaço livre entre top e $free$. Como o objeto 3 pode ser encaixado nesse espaço, seu novo endereço será a posição apontada por $free$.

O próximo objeto a ser visitado, objeto 4, também é um objeto fixo. Novamente, nenhum endereço precisa ser calculado e o apontador top é atualizado. Essa situação está ilustrada na Figura 4.1(e).

No próximo passo, o apontador h alcança o último objeto a ser visitado. Nesse ponto, o apontador $free$ está referenciando o endereço dado na Figura 4.1(e) e o apontador top acaba de ser atualizado para a posição dada pela Figura 4.1(f). Note que o objeto 5 não cabe no espaço existente entre $free$ e o próximo objeto fixo (2). Então esse espaço se torna o buraco B1, e $free$ passa apontar para a posição imediatamente após ao referido

objeto fixo. Agora, podemos mover o objeto 5 para a posição apontada por *free*. Teremos a situação ilustrada pela Figura 4.1(f).

Isso encerra o cálculo dos endereços de *forward* para todos os objetos vivos do *heap*. A Figura 4.1(g) mostra como estará o *heap* ao final de toda a fase de compactação, ou seja, após a aplicação dos Algoritmos 15 e 16. Os blocos livres que restaram abaixo do apontador *top*, nesse caso B1 e B2, foram colocados na lista de blocos livres para reutilização por parte do alocador.

Algoritmo 14 Calcula Novos Endeços - JVMPack

function Compute Address()

```
h= free = top = heap→botton
while h < heap→end
    if h points to a valid object
        if Object is fixed
            h→forward = top
            top = h + size
        else
            if Object fits in the next free position
                h→forward = free
                free = free + size
            else
                if Object fits in a hole between free and top
                    free = botton of the hole
                    h→forward = free
                    free = free + size
                else
                    h→forward = top
                    free = top = top + size
    h = h + size
```

Algoritmo 15 Atualiza Apontadores - JVMPack

function Update_Pointers()

for All objects on the black list

if $obj \rightarrow forward \neq$ NULL

for all pointers inside obj

/* t_obj_i is the target object of p_i */

$p_i = t\_obj_i \rightarrow forward$

Algoritmo 16 Move Objetos - JVMPack

function Move_Objects()

for All obj in black list

$next = obj \rightarrow next$

$prev = obj \rightarrow prev$

if ($next$ is not fixed AND $next \rightarrow forward \neq$ NULL)

$obj \rightarrow next = next \rightarrow forward$

if ($prev$ is not fixed AND $prev \rightarrow forward \neq$ NULL)

$obj \rightarrow prev = prev \rightarrow forward$

$h = heap \rightarrow botton$

while $h < heap \rightarrow end$

if h is alive AND h is not fixed

copy h to $h \rightarrow forward$

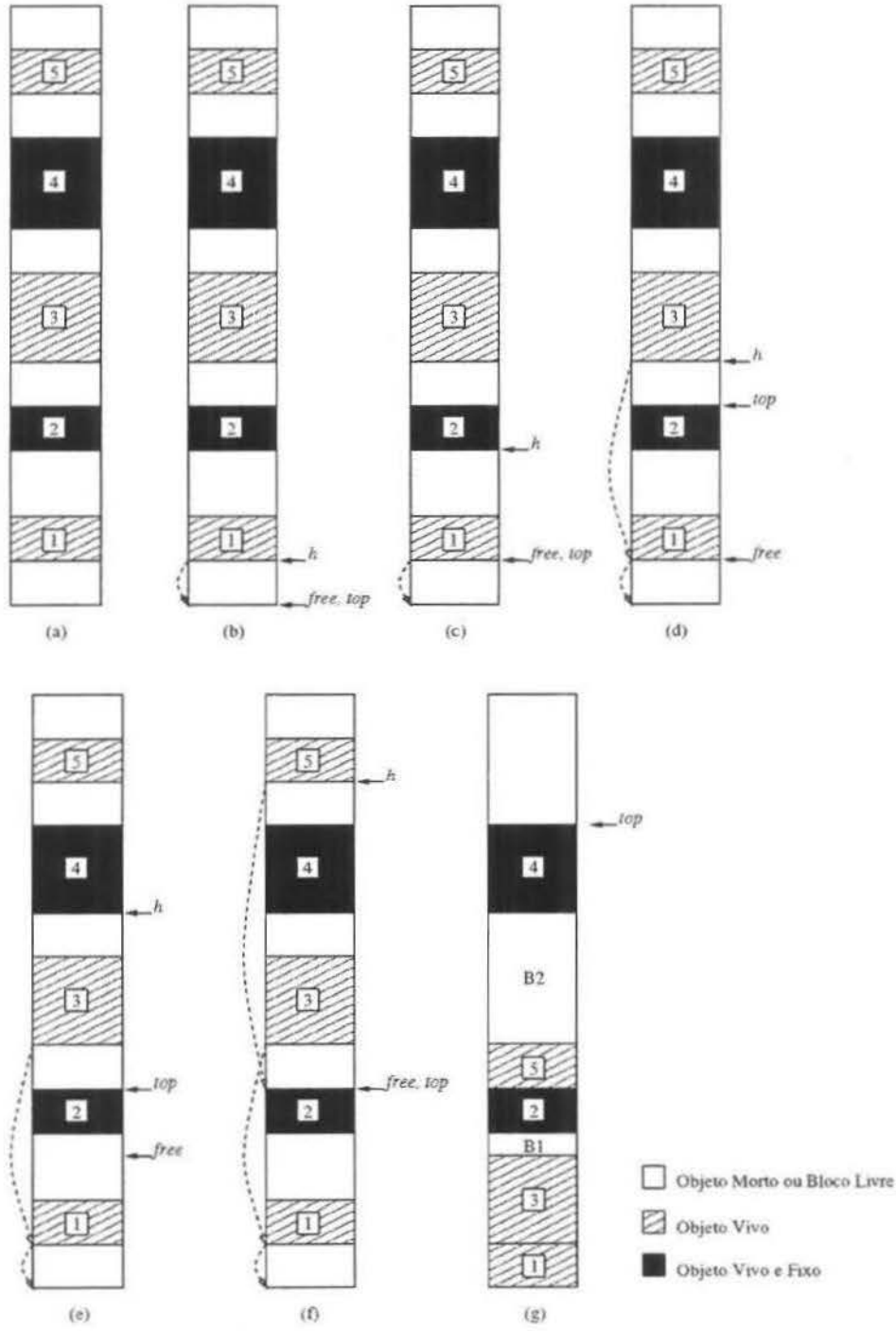


Figura 4.1: Exemplo de Fucionamento do Compactador JVMPack

Capítulo 5

Resultados Experimentais

Ao longo deste capítulo, vamos avaliar os resultados obtidos pelo sistema de gerenciamento de memória que implementamos. Inicialmente descrevemos os programas contidos no pacote de benchmarks SPEC JVM98, que foram utilizados para essa avaliação. A seguir, apresentamos os resultados obtidos, sempre que possível comparando com os valores que são obtidos pelo sistema originalmente presente em *Kaffe*.

5.1 Descrição dos Benchmarks

Escolhemos utilizar programas de benchmark presentes no pacote SPEC JVM98 por esse ser composto de várias aplicações reais implementadas na linguagem Java, que exercem uma grande demanda ao sistema de gerenciamento de memória. Os programas que foram utilizados na avaliação são os seguintes:

- *_200_check:* Esse programa testa várias características da máquina virtual Java para garantir que as execuções serão confiáveis. Esse teste inclui:
  - Indexação de vetores além de seus limites;
  - Cria super-classes e sub-classes, então acessa dados públicos, privados e protegidos e sobrescreve métodos;
  - Verifica se a JVM faz suposições sem sentido sobre *non-final* métodos. Existe uma classe e uma sub-classe que implementam um método de maneira diferente, então, se a JVM utiliza o método da super-classe, obtém-se um resultado errado;
  - Um conjunto de testes chamado PepTest, onde são verificados os seguintes aspectos: execução de *if-then-else*, operações com vetores, laços *for*, divisão, operadores bit a bit, etc.

Seu objetivo é apenas garantir que o ambiente é adequado à execução de aplicações Java, não fornecendo dados úteis à avaliação de desempenho. A máquina virtual *Kaffe* recebeu conceito *OK* desse teste rodando tanto com o coletor original quanto com o que implementamos.

- *_201_compress :* Implementação do algoritmo extraído de *A Technique for High Performance Data Compression*, Terry A. Welch, IEEE Computer Vol 17, No 6 (June 1984), pp 8-19. Este constitui-se de um método de Lemple-Ziv (LZW) modificado. Basicamente encontra sub-cadeias comuns de caracteres e as troca por códigos de tamanho variado. Durante a execução, realiza cinco iterações sobre arquivos de entrada que variam de 0.9 Mbytes a 3Mbytes, sendo que cada um é lido, comprimido, o resultado é escrito na memória e depois lido novamente, descomprimido e seu tamanho final comparado. Esse programa aloca 334Mbytes de objetos, sendo que dois grandes vetores são alocados a cada iteração, um para a entrada e outro para a saída, e muito poucos objetos além desses. Isso faz com que esse programa tenha um comportamento bem particular com relação à ocupação da memória.

- *_202_jess:* *Java Expert Shell System (JESS)* é baseado no *expert shell system* da NASA chamado *CLIPS*. De maneira geral, esse programa aplica um conjunto de instruções *if-then*, chamadas de regras, a um conjunto de dados, chamados de lista de fatos. Jess resolve uma série de problemas comumente usados com CLIPS. Esse programa aloca um total de 748 Mbytes de objetos, mas como não armazena resultados temporários por muito tempo, seu *heap* vivo se mantém relativamente constante ao longo da execução.

- *_209_db:* Realiza múltiplas funções em um banco de dados residente na memória. Lê um arquivo de entrada com um tamanho de 1Mbyte, contendo nomes, endereços e telefones. Lê também um outro arquivo, chamado *scr6*, que contém as ações a serem executadas nos registros do arquivo. Dentre os comandos executados estão: adicionar, apagar, procurar endereco e ordenar endereços. Esse programa aloca 224 Mbytes de objetos, possuindo um comportamento bem particular: o tamanho do *heap* cresce linearmente durante a construção do banco de dados e permanece em um tamanho fixo ao longo do resto da execução.

- *_227_mtrt:* Essa é uma variante de um benchmark conhecido como _205_raytrace, um *raytracer* que trabalha numa cena mostrando um dinossauro, onde duas *threads* renderizam a cena contida no arquivo de entrada modelo *time-tes*. Esse programa aloca 355 Mbytes de objetos, e é o único com mais de uma *thread* presente no pacote.

- *_228_jack:* Jack é um gerador de parser Java baseado na ferramenta *Purdue Compiler Construction Tool Set (PCCTS)*. Essa é uma antiga versão do programa atualmente conhecido como Javacc. Um arquivo *jack.jack* contém instruções para que o Jack gere a si mesmo. Esse arquivo é fornecido ao programa para que este seja gerado múltiplas vezes. Essa aplicação aloca 481 Mbytes de objetos e, semelhantemente ao JESS, seu *heap* não cresce de acordo com a grande quantidade total de dados alocada porque esses dados não sobrevivem entre iterações.

Esse programas podem ser executados com um parâmetro -sX, onde X indica o tamanho da entrada. Os dados sobre a alocação de objetos apresentados na descrição acima são relativos à $X = 100$, indicando 100% de carga de trabalho. O parâmetro X também pode assumir os valores 1 e 10, indicando que o programa será executado com, aproximadamente, 1% e 10% da carga de trabalho.

Os dados que constam na descrição acima foram retirados da documentação fornecida pela SPEC. Dieckmann e Hölzle apresentam em [18] dados bem diferentes quanto à quantidade total de memória alocada em cada programa do pacote. Os dados apresentados nesse trabalho podem ser vistos na Tabela 5.1.

Programa	Heap Vivo Máximo (Mbytes)	Alocação Total (Mbytes)
Compress	6.7	105
DB	7.2	231
Jack	1.2	61
Jess	1.1	161
Mtrt	3.0	147

Tabela 5.1: Dados de Alocação para os Benchmarks do Pacote SPEC JVM98

5.2 Análise dos Resultados

Nosso sistema de recuperação de memória, como descrito no início do Capítulo 4, possui uma certa liberdade em sua configuração. Isso se dá com respeito à escolha do parâmetro indicador de quanto em quanto tempo será realizada a compactação.

Vários experimentos foram realizados variando esse parâmetro, visando determinar qual a configuração que possui melhor desempenho. À *priori*, espera-se que um maior número de compactações acarrete um melhor aproveitamento da memória disponível e, devido ao esforço extra, diminua um pouco o desempenho da aplicação. Essa última colocação pode deixar de ser verdadeira se a melhora no aproveitamento do espaço no

heap for tão boa a ponto de reduzir o número de coletas necessárias durante toda a execução, compensando o esforço extra realizado.

Infelizmente, devido à presença de objetos fixos, o comportamento de nosso sistema não foi tão estável quanto a essa previsão. Algumas configurações que deveriam ter um melhor uso da memória, não o tiveram devido à influência dos objetos fixos, principalmente no momento da alocação de um objeto grande.

Outro parâmetro que pudemos variar na execução dos benchmarks foi a sua carga de trabalho. Executamos esses programas com 10% e com 100% de sua carga, valores com os quais os programas apresentaram um uso razoável do sistema de recuperação de memória.

Em nosso laboratório, tivemos duas plataformas disponíveis para testes de nosso sistema. São elas:

1. SPARC, rodando o sistema operacional Solaris versão 7. Essa máquina possui 4 Gbytes de memória RAM física e 4 processadores de 400 Mhz, sendo que cada processo fica alocado a apenas um deles.

2. AMD K6-2, rodando o sistema operacional Linux, distribuição RedHat versão 6.0 com kernel versão 2.2.5-15. Essa máquina possui 64 Mbytes de memória RAM física e um processador de 350 Mhz.

A seguir, apresentamos os resultados obtidos para cada uma dessas plataformas.

5.2.1 Solaris

10% da Carga de Trabalho

Para avaliar o sistema rodando os benchmarks sob essa carga de trabalho, utilizamos dois esquemas. O primeiro realiza compactação a cada 3 coletas. No segundo, realizamos uma heurística para tentar melhorar o aproveitamento da memória disponível. Nesse caso a compactação é realizada obrigatoriamente a cada 5 coletas, ou toda vez que o coletor identificar que uma coleta tradicional não será suficiente para recuperar um bloco de memória do tamanho necessário. Tal previsão é possível pois sabemos quanto de memória foi requisitado e não encontrado pelo alocador e, ao final da fase de marcação, todo objeto que estiver na lista de brancos está morto, portanto, basta verificar se há bloco de tamanho suficiente nessa lista. Se tal bloco não existir, fazemos a compactação tentando evitar uma requisição de memória desnecessária ao sistema operacional.

A Tabela 5.2 mostra os resultados obtidos utilizando-se o primeiro esquema de compactação. A Tabela 5.3 compara os resultados obtidos utilizando a heurítica citada acima. Nessas tabelas, a coluna `Org` representa o coletor original e a coluna `Imp`, o coletor implementado nesse projeto. Por esses dados vemos que o segundo esquema se mostra mais

eficiente, tornando nosso sistema mais rápido do que o original em dois casos: Jess e Compress.

Programa	Tempo de Execução (s)			
	Melhor		Médio	
	Org	Imp	Org	Imp
Jess	42.060	43.985	42.114	44.573
Jack	78.052	94.808	78.056	96.608
Compress	251.827	251.639	251.914	252.504
Mtrt	115.529	176.637	115.556	181.912
DB	20.398	23.098	20.424	23.260

Tabela 5.2: Comparação entre os coletores para entradas de tamanho 10

Programa	Tempo de Execução (s)			
	Melhor		Médio	
	Org	Imp	Org	Imp
Jess	42.060	41.916	42.114	41.986
Jack	77.951	88.648	78.056	90.704
Compress	251.827	243.111	251.914	243.341
Mtrt	115.529	198.461	115.556	205.481
DB	20.398	23.107	20.424	24.745

Tabela 5.3: Comparação entre os coletores para entradas de tamanho 10 - Esquema II

100% da Carga de Entrada

Apresentaremos agora um retrato mais detalhado da execução dos benchmarks considerando sua carga total de trabalho.

A Tabela 5.4 mostra os valores obtidos utilizando-se o sistema de gerenciamento de memória original da JVM *Kaffe*. As Tabelas 5.5 e 5.6 apresentam os melhores resultados obtidos pelo nosso sistema, com relação ao tempo de execução e consumo de memória, respectivamente. Nessas tabelas, a letra H na coluna de Frequência de Compactação, indica o uso da mesma heurística aplicada anteriormente, visando melhorar o consumo de memória. Vemos que essa heurística nos dá a melhor utilização da memória em três casos sendo que, dois deles também representam o melhor tempo de execução.

O comportamento dos programas conforme variamos a frequência das compactações, com relação ao tempo de execução e consumo de memória, está ilustrado pelos gráficos contidos nas Figuras 5.1 e 5.2. Através desses gráficos podemos ver as ocilações no comportamento causadas pela presença dos objetos fixos. No gráfico 5.2, existem alguns pontos com maior frequência de compactação que não proporcionaram um melhor uso da memória e, por sua vez, existem pontos com compactação não tão frequente que obtiveram melhores desempenhos nesse aspecto. O resultado inesperado mais flagrante quanto a esse aspecto é o do programa Compress. Na seção 5.2.3, apresentamos uma justificativa para esse comportamento.

Programa	Tempo de Execução (s)	Tamanho Máximo do Heap (KBytes)
Jess	872.77	6144
Jack	657.89	9120
Compress	2918.35	28492
Mtrt	1177.40	17407
DB	1234.16	17407

Tabela 5.4: Resultados para o Coletor Original - Solaris

Programa	Tempo de Execução (s)	Tamanho do Heap (KBytes)	Freq. Comp.
Jess	871.22	5120	H
Jack	750.09	5120	H
Compress	2830.63	35204	4
Mtrt	1463.70	17407	5
DB	1364.32	22528	6

Tabela 5.5: Melhores Resultados para Tempo de Execução - Solaris

Programa	Tempo de Execução (s)	Tamanho do Heap (KBytes)	Freq. Comp.
Jess	871.22	5120	H
Jack	750.09	5120	H
Compress	2912.51	20904	0
Mtrt	1571.49	16384	1
DB	1494.26	17407	H

Tabela 5.6: Melhores Resultados para Consumo de Memória - Solaris

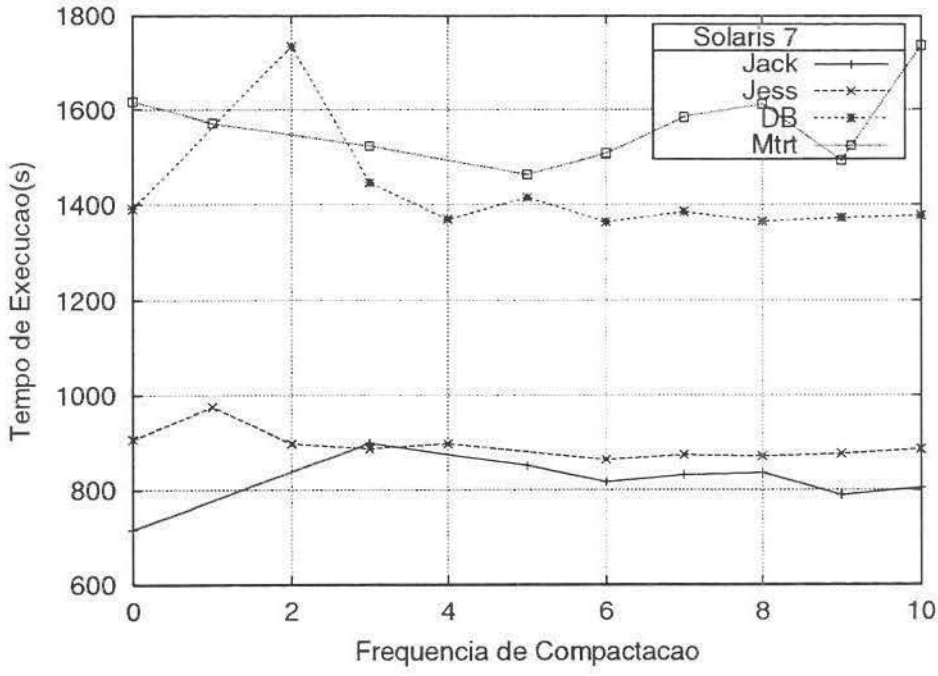


Figura 5.1: Tempo de Execução - solaris

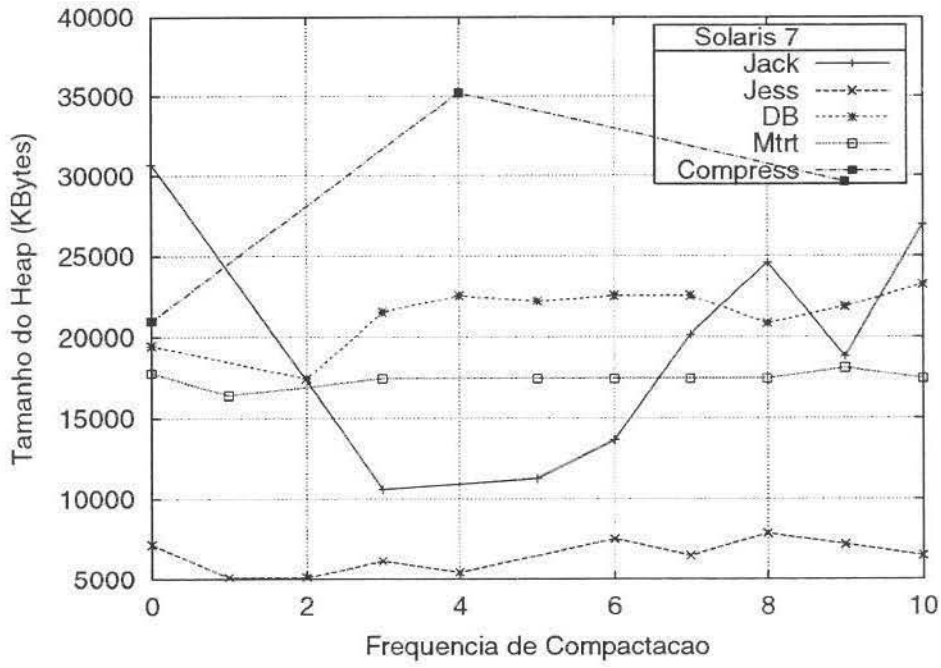


Figura 5.2: Consumo de Memória - solaris

5.2.2 Linux

Nessa seção constam os resultados obtidos executando-se os benchmarks na plataforma Linux RH6.0. Essa máquina possui a característica de ser mono-usuário durante todo o tempo, fator que não pudemos assegurar para a plataforma anterior, apesar de termos executado os testes garantindo sempre um processador exclusivamente dedicado ao nosso processo.

Não existem diferenças drásticas quanto ao comportamento em relação à plataforma Solaris. Podemos ver pela Tabela 5.8 que a maioria dos programas obteve seus melhores resultados sob a mesma configuração usada na máquina Solaris. A pricipal diferença, que pode ter surgido pelo fato de ser uma máquina garantidamente mono-usuário, é que as mesmas configurações apresentaram os melhores resultados tanto para tempo de execução como para consumo de memória sendo que, para o Solaris, três benchmarks obtiveram seus melhores resultados em configurações diferentes com relação a esses dois aspectos.

Programa	Tempo de Execução (s)	Tamanho Máximo do Heap (KBytes)
Jess	1195.07	6144
Jack	1045.02	9120
Compress	4725.26	28492
Mtrt	1623.02	17407
DB	2090.41	17407

Tabela 5.7: Resultados para o Coletor Original - Linux

Programa	Tempo de Execução (s)	Tamanho Heap (KBytes)	Freq. Comp.
Jess	1184.28	5120	H
Jack	1031.03	5120	H
Compress	4136.50	17884	0
Mtrt	1960.77	17407	5
DB	2182.20	19456	1

Tabela 5.8: Melhores Resultados com 100% de Carga de Trabalho - Linux

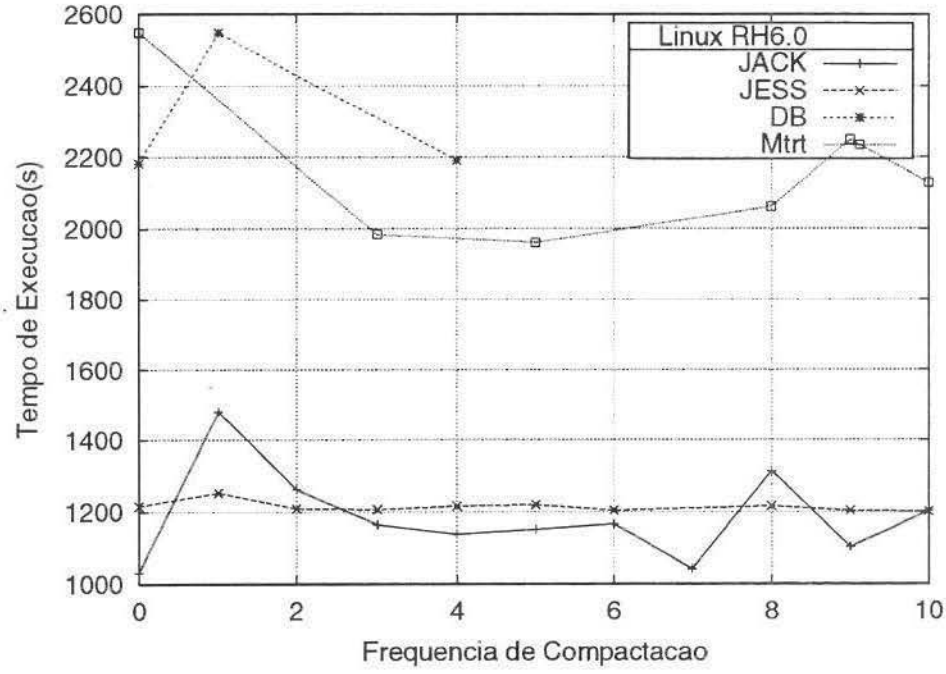


Figura 5.3: Tempo de Execução - Linux

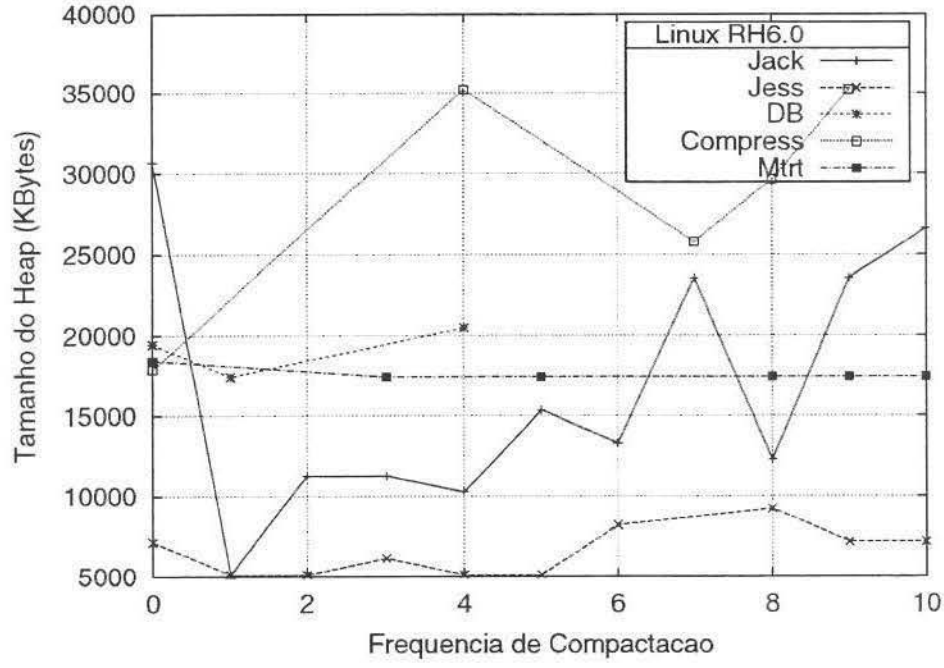


Figura 5.4: Consumo de Memória - Linux

5.2.3 Considerações Gerais

Analisando os gráficos e tabelas referentes às duas plataformas onde foi testado nosso sistema, podemos verificar que os programas onde obtivemos os melhores resultados, com relação ao consumo de memória, foram Jess e Jack. O programa Mtrt nos trouxe resultados bastante parecidos com o coletor original em termos de consumo de memória, mas apresentou uma perda considerável em termos de tempo de execução, em torno de 20% em ambas as plataformas. Esse programa trabalha a maior parte do tempo com o *heap* próximo de sua capacidade total, isso ocasiona muitas coletas que acabam recuperando pouca memória devido à grande quantidade de objetos vivos. Isso faz com que o esforço das compactações aumente o tempo de execução e não obtenha um grande ganho no consumo de memória.

As Figuras de 5.5 a 5.7 apresentam gráficos que ilustram a quantidade de objetos fixos presentes no *heap* ao longo da execução de cada benchmark. O programa Jess possui uma quantidade de objetos fixos praticamente constante ao longo de sua execução, sendo que uma grande parte esta localizada na extremidade inferior do *heap*, prejudicando menos a compactação. Conforme a execução caminha, o número de objetos alocados aumenta, fazendo que a proporção de objetos fixos no *heap* diminua consideravelmente após um certo número de coletas. Este é o programa que apresentou os melhores resultados em nosso sistema. O gráfico da Figura 5.6 corresponde ao período referente a uma das dezesseis repetições realizadas pelo programa, sendo que esse padrão se repete para cada uma delas, pois os objetos não sobrevivem ao final de cada execução. Esse programa não possui uma taxa tão baixa de objetos fixos, estando na maior parte do tempo em torno de 20% mas, possivelmente pelo fato de não sobreviverem durante muito tempo, não causam grande prejuízo ao nosso sistema de compactação visto que, nesse caso, o consumo de memória foi melhor que o do sistema original para algumas configurações. O programa Mtrt é o que possui a menor proporção de objetos fixos no *heap*, pois grande parte de sua alocação corresponde a objetos instanciados [18], os quais normalmente podem ser movidos. O programa DB mantém uma taxa de objetos fixos constante em torno e 42%. Essa é a maior taxa entre os benchmarks utilizados e permanece constante após algumas poucas coletas devido à característica apontada na descrição dada anteriormente: esse programa aloca uma grande quantidade de objetos no começo e depois realiza as operações de bancos de dados sobre esses dados, não executando grande volume de alocação ao longo da execução.

Vamos utilizar o programa Compress para ilustrar um efeito prejudicial ao nosso sistema de compactação causado pela presença de objetos fixos. Esse problema é o responsável pelo consumo exagerado de memória tido por esse programa quando utilizamos compactação. Esse programa tem uma característica bem peculiar com respeito à alocação de memória: ele aloca dois grandes vetores a cada iteração e poucos objetos pequenos du-

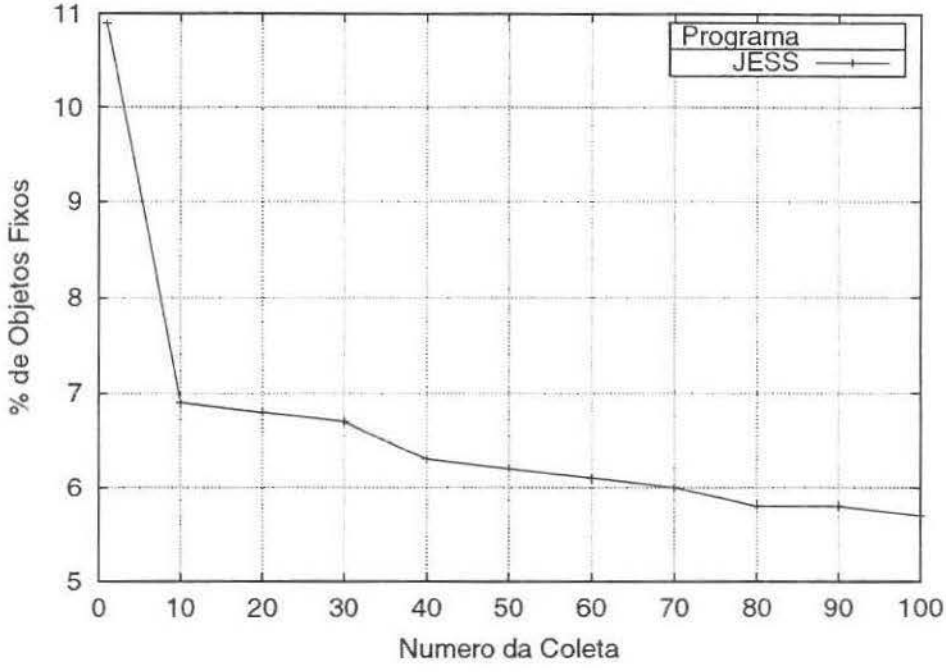


Figura 5.5: Porcetagem de Objetos Fixos - Jess

rante a sua execução. Isso faz com que ao início de cada iteração do algoritmo para cada arquivo de entrada seja necessária uma grande quantidade contínua de memória livre. O uso de compactação deveria auxiliar nesse aspecto, mas devido à presença de objetos fixos em posições estratégicas no *heap* esse objetivo não é atingido. A Figura 5.8 ilustra uma situação encontrada no *heap* durante uma execução: suponha que o programa solicite um espaço de 3Mbytes ao alocador e que não existe tal quantidade de memória disponível. Suponha ainda que essa é uma coleta programada para utilizar compactação. A configuração do *heap* antes da coleta mostra que existe um bloco de objetos mortos de tamanho suficiente e logo a seguir um objeto fixo. Após a realização da compactação vemos que a presença do objeto fixo impediu a criação de um espaço livre contínuo ainda maior, e fez com que não existisse mais um bloco com o tamanho necessário. Dessa forma a única saída para o alocador é requisitar mais um bloco de 3Mbytes ao sistema operacional. Situações como essa são a causa das ocilações no gráfico de consumo de memória, pois dependem de quando as compactações estão sendo realizadas.

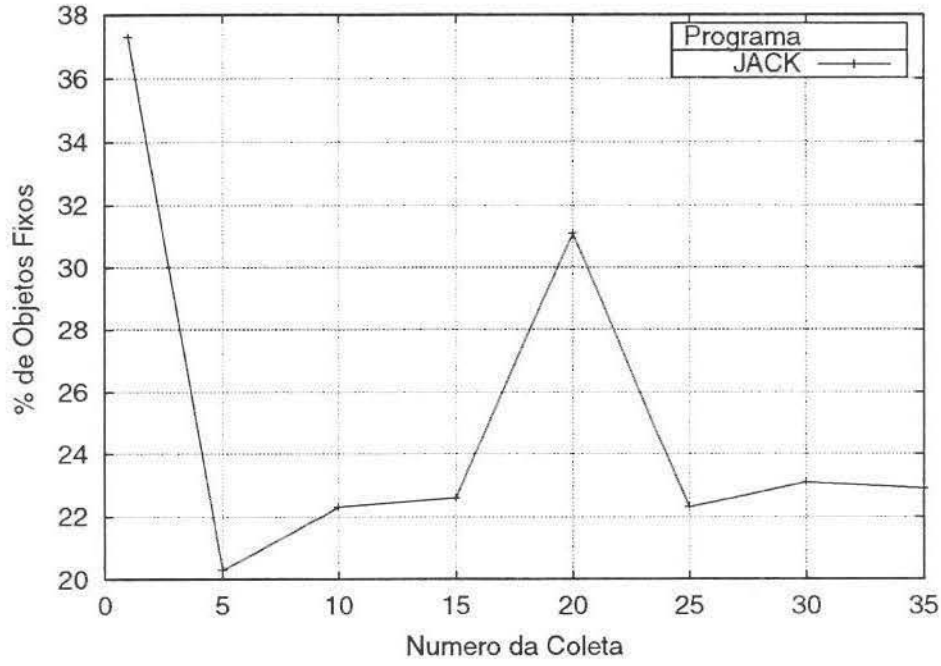


Figura 5.6: Porcetagem de Objetos Fixos - Jack

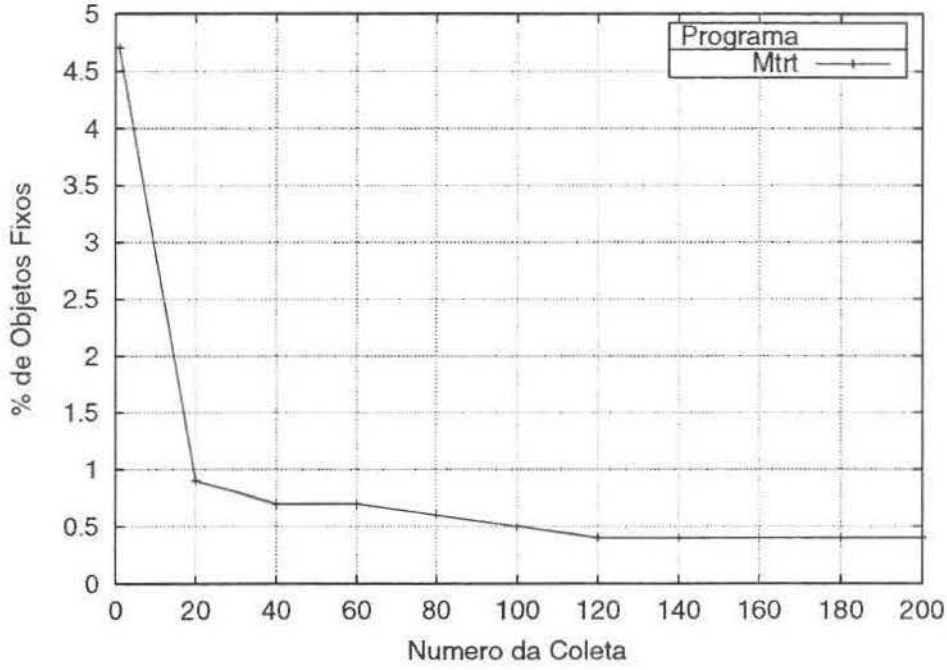


Figura 5.7: Porcetagem de Objetos Fixos - Mtrt

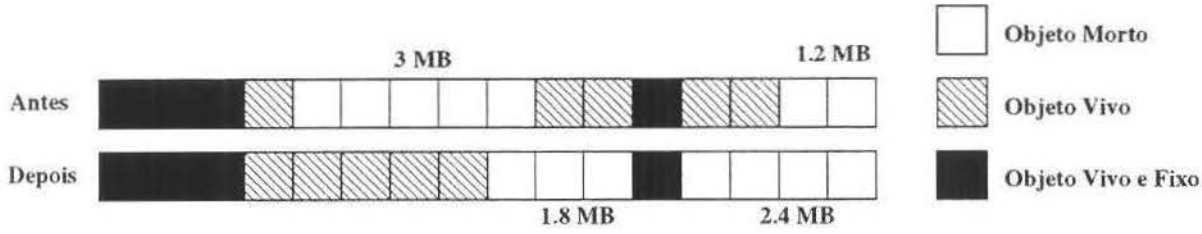


Figura 5.8: Efeito dos Objetos Fixos no Programa Compress

Capítulo 6

Conclusão e Trabalhos Futuros

Nesse último capítulo tiramos algumas conclusões baseadas nesses dados e no acompanhamento do comportamento dos objetos no *heap* durante a execução, apontando como o desempenho poderia ser melhorado com algumas modificações em pontos da máquina virtual que são externos ao sistema de gerenciamento de memória e que, portanto, não foram realizadas por fugirem do escopo desse projeto. Essas sugestões de modificação são destacadas entre os possíveis trabalhos futuros ao final do capítulo.

6.1 Principais Problemas e Possíveis Soluções

O principais problemas existentes na JVM *Kaffe* para a implementação de um coletor que movimente os objetos no *heap* são causados pela falta de informações fornecidas ao coletor pelo compilador. Isso obriga a implementação de um coletor conservativo. Objetos fixos são causados pela falta de identificação dos ponteiros nas pilhas de execução das *threads*.

Uma solução para os ponteiros nas pilhas de execução pode ser a adoção de *Tagged Pointers*. Como mencionado na Seção 4.2.3, essa abordagem insere bits de identificação antes de cada campo nas pilhas de execução, tornando-os auto-descritivos.

Uma outra solução para esse problema pode ser aquela descrita por Dieckmann e Hölzle, em seu estudo sobre alocação de memória nos programas do pacote SPECjvm98 [18], no qual eles afirmam que a máquina virtual presente no JDK1.1.5 da Sun utiliza um tipo de apontador, chamado *handle*, para possibilitar ao coletor a realização da compactação dos objetos. Esse apontador se encaixa no conceito de *forwarding pointer*, isto é, os objetos não são referenciados diretamente, mas sim, referenciados somente pelo seu respectivo *handle*. Isso implica que, para alterar todas as referências a um determinado objeto, precisamos apenas atualizar seu *handle* com o novo endereço. Esses apontadores *handle* podem ser armazenados numa região particular do *heap*, evitando que atrapalhem a compactação e, como permanecem no mesmo endereço durante toda a vida do objeto,

constituem um identificador ideal para o mesmo.

Existem trabalhos visando tornar o coletor da máquina virtual Java exato. Alguns desses trabalhos estão descritos na Seção 1.2. Tornando o coletor da máquina virtual *Kaffe* exato estaríamos não só proporcionando uma compactação total do *heap*, mas também acelerando as coletas em geral, pois eliminaríamos a tarefa de tentar identificar possíveis apontadores conservativamente. Resultaria também num melhor aproveitamento da memória disponível, pois não haveria o risco de reter na memória objetos marcados equivocadamente. Algumas dessas técnicas conseguem inclusive reduzir o número de raízes, detectando quais delas já estão mortas no ponto onde foi acionado o coletor.

Um outro fator que contribue para a degradação do uso eficiente da memória é o alinhamento dos objetos. A Tabela 6.1, com dados extraídos de [18], mostra que o *heap* é basicamente dominado por objetos de tamanho pequeno, em todos os benchmarks utilizados, à exceção do Compress.

	Objetos Instanciados		
Programa	**Média**	**Mediana**	**% do Total**
Compress	17	16	0
DB	12	12	58
Jack	17	16	45
Jess	23	24	51
Mtrt	16	16	74

Tabela 6.1: Tamanho Médio em Bytes dos Objetos Alocados

Como dito na seção 4.1.1, nosso alinhamento é baseado em unidades do cabeçalho, isto é, todo objeto deve ter um tamanho múltiplo de 24 bytes. Com isso, temos que o menor tamanho de objeto possível em nosso sistema é de 48 bytes, correspondentes a uma unidade para o próprio cabeçalho e outra para acomodar o objeto em si. Isso causa uma perda inevitável de memória na alocação de objetos com tamanho inferior a 24 bytes. Para ilustrar esse problema leve em consideração os tamanhos de objetos listados na Tabela 6.1, todos esses objetos ocupariam 48 bytes em nosso *heap*.

6.2 Conclusão Final

A introdução de um esquema de compactação ao sistema de gerenciamento de memória da máquina virtual *Kaffe* pode trazer melhorias ao seu desempenho. Isto porque, mesmo considerando todas as restrições impostas ao nosso sistema devido à impossibilidade de realizar, em tempo hábil, todas as alterações necessárias à máquina virtual, conseguimos

obter um desempenho similar, em alguns casos até ligeiramente superior, ao coletor original. No entanto, a viabilidade de implantação de um sistema como esse fica certamente condicionada à realização de tais alterações para que seja possível a compactação completa do *heap*, isto é, eliminando a existência de objetos fixos. Mais uma evidência é o fato de a *Kaffe*, mesmo com seu coletor original, utilizar *heaps* de até 20 MBytes, enquanto em [18] os autores afirmam que todos os programas presentes no pacote SPECjvm98 podem rodar com um *heap* de no máximo 8 MBytes. A presença de objetos fixos em pontos estratégicos, como descrito para o programa *Compress* no capítulo anterior, impossibilitaram nosso sistema de se aproximar desse resultado em alguns casos, mesmo fazendo uso da compactação.

A solução descrita como *handle pointers* nos parece bastante interessante. Isso porque resolve o problema dos objetos fixos e ainda diminui o trabalho na atualização dos apontadores, pois serão os únicos a serem alterados nessa fase. Aparentemente, o principal obstáculo à sua implementação será resolver o problema de como deixar a máquina virtual ciente de que, toda referência sua a um objeto no *heap* é, na verdade, uma referência à uma referência ao objeto. Entretanto, a solução ideal seria tornar o coletor exato, isto é, combinar técnicas estáticas e dinâmicas como as descritas na seção 1.2, para que o compilador forneça todas as informações necessárias a esse tipo de coletor. Com certeza essa solução implicaria em profundas alterações em várias áreas da máquina virtual.

6.3 Trabalhos Futuros

Baseados nessas conclusões, podemos sugerir os seguintes temas para futuros estudos sobre o comportamento do sistema de gerenciamento de memória da máquina virtual Java com a implantação de um algoritmo de compactação:

- Implementar uma das soluções acima citadas para viabilizar a obtenção de informações sobre os apontadores presentes nas pilhas de execução, possibilitando ao coletor identificá-los precisamente e, principalmente, alterá-los conforme a movimentação dos objetos. O mesmo procedimento vale para os eventuais apontadores externos ao *heap* presentes no conjunto de raízes que, como vimos, é representado por uma $hash - table$ na JVM *Kaffe*.

- Estudar a viabilidade de adoção de outros esquemas de alocação, tentando diminuir o efeito causado pela adoção de objetos com tamanho múltiplo de 24 bytes.

- Adaptar o sistema implementado ao coletor da máquina virtual *Kaffe* versão 1.0.5. Uma avaliação superficial mostrou que existem várias mudanças na versão atual do

sistema de recuperação de memória, certamente acompanhadas de uma melhora de desempenho.

- Estender o sistema para operar com o *JIT(Just in Time Compiler)*.

Visando colaborar com a realização desses possíveis trabalhos, estamos disponibilizando uma versão *postscript* dessa dissertação, bem como o código implementado para nosso sistema no url `http://www.ic.unicamp.br/~lsc/Projects/GCKaffe.html`.

Referências Bibliográficas

[1] Ole Agesen and David Detlefs. Finding References in Java Stacks. Technical report, Sun Microsystems Laboratories, 1997.

[2] Ole Agesen, David Detlefs, and J.E.B. Moss. Garbage Collection and Local Variables Type-Precision and Liveness in Java Virtual Machines. In *In proceedings of PLDI'98*, pages 269–279, Montreal, May 1998.

[3] S.B. Akers. Binary Decision Diagrams. *IEEE Trans. Comput.*, C-26(6(Aug)):509–516, 1978.

[4] A. Appel. *Modern Compiler Implementation in Java - Basic Techniques.* Cambridge University Press, 1997.

[5] A. Hanson D.R. Appel. Copying Garbage Collection in the Presence of Ambiguous References. Technical report, Princeton University, 1988. CS-TR-162-88.

[6] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1998.

[7] Henry G. Baker Jr. List processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[8] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical report, Western Research Laboratory - Digital Equipment Corporation, 1988.

[9] M. Boehm, H. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[10] Z. Boehm, H. Shao. Infering Type Maps During Garbage Collection. *OOPSLA/ECOOP'93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1988.

[11] R.E. Brace, K.S. Bryant and R.L. Rudell. Efficient Implementation of a BBD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, 1990.

[12] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *Communications of the ACM*, 24(3):293–318, 1992.

[13] E.M. Burch, J.R. Clarke and K. McMillan. Symbolic Model Checking: 10^{20} States and Beyond. In *The 5th Annual IEEE Symposium on Logic in Computer Science.*, pages 428–439, Philadelphia, 1990.

[14] C.J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[15] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, 1960.

[16] C.E. Cormen, T.H. Leisersonb and R.L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[17] J.C. Coudert, O. Madre and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building Their State Diagrams. *Computer-Aided Verification '90*, pages 75–84, 1990.

[18] S. Dieckmann and U. Holzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. Technical Report 33, University of California at Santa Barbara, 1998.

[19] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[20] J.C. Fenichel, R.R. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, 1969.

[21] James Gosling. Java Intermediate Bytecodes. In *Proceedings of the ACM SIGPLAN '95 Workshop on Intermediate Representations (IR'95)*, pages 111–118, January 1995.

[22] R. Jones and R. Lins. *Garbage Collection: Algorithm for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[23] D.E. Knuth. *The Art of Computer Programming, vol I: Fundamental Algorithms.* Addison-Wesley, 1967.

[24] Tim Lindholm and Yellin Frank. *The Java Virtual Machine Specification.* Addison-Wesley, 1998.

[25] R. Lins. Cyclic Reference Counting with Lazy Mark-Scan. *Information Processing Letters*, 4(44):215–220, 1992.

[26] A.D. Martinez, R.D. Lins, and Wachenchauzer R. Cyclic Reference Counting with Local Mark-Scan. *Information Processing Letters*, (34):31–35, 1990.

[27] John McCarthy. Recursive Functions of symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3:184–195, 1960.

[28] Alexandre Petit-Bianco. Java Garbage Collection for Real-Time Systems. *Dr. Dobb's Journal*, (10):20–29, 1998.

[29] J.M. Stichnoth, Guei-Yuan Lueh, and Michal Cerniak. Support for Garbage Collection at Every Instruction in a Java Compiler. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language Design and Implementation*, pages 118 – 127, 1999.

[30] J. Stolfi. Métodos Automáticos de Gerência de Memória. Master's thesis, Instituto de Matemática e Estatística, Universidade de São Paulo, 1979.

[31] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. Technical report, University of Texas, 1994. Expanded Version of the IWMM92 paper in ACM Computing Surveys.