Coleta de Lixo na JVM

Adriano da Silva Castro

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Orientador: Prof. Marcelo Lobosco

Juiz de Fora, MG
Dezembro de 2009

Coleta de Lixo na JVM

Adriano da Silva Castro

Monografia submetida ao corpo docente do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora, como parte integrante dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Aprovada pela banca constituída pelos seguintes professores:

_____

Prof. Marcelo Lobosco – Orientador
DSc. em Engenharia de Sistemas e Computação,
COPPE/UFRJ, 2005

_____

Prof. Jairo Francisco de Souza
M.Sc em Engenharia de Sistemas e Computação
COPPE/UFRJ, 2007

_____

Prof. Eduardo Barrére
DSc. em Engenharia de Sistemas e Computação
COPPE/UFRJ, 2007

Juiz de Fora, MG
Dezembro de 2009

Agradecimentos

Agradeço, principalmente, a meus pais, que sempre estiveram ao meu lado com total apoio, sempre apoiando minhas decisões, incondicionalmente.

Agradeço à minha namorada Lívia pelo amor, carinho e companheirismo único ao longo dessa reta final; e ao curso, por permitir que eu a conhecesse.

Agradeço a meus parentes, à Polly e a todos(as) os(as) meus(as) companheiros(as) de curso pelas festas, apoio e por serem um exemplo de união ao longo destes 4 anos.

Agradeço ao prof. Marcelo Lobosco pelas horas perdidas comigo e pela dedicação fora do comum durante o desenvolvimento deste trabalho.

Por fim agradeço a todos os companheiros de república que tive durante os anos de curso e que foram mais do que amigos... foram irmãos.

Sumário

Lista de Figuras

Resumo

A coleta de lixo na memória é um recurso disponível em diversas linguagens e surgiu da necessidade de garantir que o lixo presente na memória seja sempre reciclado, tirando a responsabilidade do programador e atuando de forma automatizada. E por esse fator, os coletores são construídos visando uma execução transparente e otimizada. Este trabalho descreve os principais tipos de coletores e como eles são aplicados à JVM (Java Virtual Machine), abordando, também, algumas possíveis estratégias de otimização; e para concluir, um simulador implementado em Java contendo rotinas presentes na máquina virtual é o responsável por analisar três dos principais algoritmos e seus desempenhos.

Abstract

Garbage collector is a feature available in many languages and was developed by the need to guarantee that the trash elements in memory will always be recycled, taking away the programmer's responsibility by acting automatically. The collectors are implemented aiming a transparent and optimized execution. This paper describes the main collector types and how they are applied in JVM (Java Virtual Machine). Possible optimizing strategies are discussed; and as a conclusion, a simulator implemented in Java containing routines like in the virtual machine is responsible for analyzing three of the main algorithms and their accomplishment.

1 Introdução

Gerenciamento de memória é o termo utilizado, em computação, para se referir às técnicas de manutenção de memória com a finalidade, em linhas gerais, de alocar blocos de endereçamento para os processos em execução, assim como efetuar sua liberação, quando estes não estão sendo mais utilizados. O termo lixo é utilizado para denominar as regiões de memória que não são mais utilizadas e que, portanto, podem ser liberadas. O termo limpeza se refere à ação de eliminar os objetos considerados lixo na memória. O gerenciamento pode ser feito tanto de modo manual, sendo assim de inteira responsabilidade do programador, ou de forma automática, através de algoritmos que gerenciam todos os passos relacionados à alocação e a liberação da memória.

Com o crescente avanço dos microprocessadores, abre-se a possibilidade de desenvolvimento de novas aplicações, cujas demandas não podiam ser atendidas pelas gerações anteriores de processadores. Estas novas aplicações, entretanto, não demandam apenas processadores mais rápidos: muitas vezes demandam também maiores quantidades de memória. A grande quantidade de memória demandada, por sua vez, pode levar à falhas no processo de gerenciamento manual de memória, visto que o programador deve ter controle sobre um número maior de regiões de memória. Neste cenário, os mecanismos de gerenciamento automático de memória ganham destaque por eliminarem os problemas decorrentes do gerenciamento incorreto da memória. Entretanto, a comodidade do gerenciamento automático de memória cobra o seu preço na forma de incremento no tempo de computação necessário para a execução das aplicações. Assim, faz-se necessário o emprego de algoritmos otimizados e capazes de executarem a limpeza da memória com um menor custo.

Posto esse cenário, é possível introduzir o termo Coleta de Lixo. Este se refere às ações e heurísticas de liberação automática de blocos de memória ocupados por dados não mais utilizados por um determinado processo. Esses dados, em linguagens orientadas a objetos, estão codificados na forma de objetos. Linguagens como Java e .NET possuem a figura do coletor de lixo.

É interessante ressaltar que existem diversos algoritmos de coleta de lixo, com as mais distintas características. Desta forma, e tendo em vista o impacto no desempenho das aplicações decorrente da utilização de um coletor de lixo, deve-se avaliar, para cada aplicação, qual é o coletor de lixo mais adequado e que causa menor impacto em seu tempo de execução.

1.1 Motivação

Têm-se, então, dois fatores importantes e que servem de motivação para este trabalho: a crescente necessidade, por parte das aplicações, de utilização de um coletor de lixo e a queda de desempenho da aplicação resultante da utilização deste mecanismo.

1.2 Objetivos

O objetivo deste trabalho é estudar os mecanismos de coleta de lixo em determinados cenários, analisando, principalmente, o impacto de sua utilização no tempo de execução de uma aplicação. Para atingir este objetivo, procuraremos em um primeiro momento analisar o desempenho de um dos algoritmos mais utilizados pelos coletores de lixo, o *mark-and-sweep*, no escopo da linguagem de programação Java, procurando identificar seus gargalos de desempenho. Proporemos então algumas modificações no algoritmo, de forma a tentar torná-lo mais eficiente em alguns cenários.

Para dar suporte ao trabalho, um simulador de coleta foi desenvolvido. O simulador implementa não só alguns dos algoritmos de coleta já conhecidos como também a modificação proposta no algoritmo *mark-and-sweep* [BOEHM, *et al*., 1988]. O objetivo do simulador é permitir um maior controle sobre o processo de coleta, permitindo a comparação dos tempos de execução dos algoritmos em diferentes cenários de uso de memória por parte das aplicações.

1.3 Organização

Inicialmente apresentaremos um breve estudo do funcionamento dos coletores de lixo,

mais especificamente do coletor presente na Máquina Virtual Java. Em seguida, descreveremos o algoritmo de coleta de lixo escolhido para ser modificado neste trabalho, o m*ark-and-sweep*, e analisaremos o seus custos. Por fim, serão apresentados os impedimentos à sua implementação e, para justificar os algoritmos presentes na JVM, será construído um simulador ao final do trabalho.

2 Coleta de Lixo na Java Virtual Machine

Nas linguagens de programação procedurais e até mesmo em algumas que seguem o paradigma da orientação a objetos, como C++, tem-se que a limpeza de memória é uma ação controlada pelo desenvolvedor, o que significa que é de sua responsabilidade gerenciar a alocação e liberação da memória.

Essa preocupação gerada durante o desenvolvimento das aplicações é negativa, visto que uma importante parte do tempo de implementação é utilizada no gerenciamento da memória, de forma a garantir o bom funcionamento da aplicação.

Contudo, algumas linguagens automatizam o processo de limpeza de memória, eliminando o que é considerado lixo de memória sem que sejam necessárias ações por parte do programador, poupando-lhe esforços, garantindo uma alocação de memória mais eficiente e, principalmente, evitando falhas [MORAES, 2007]. Essas falhas podem ser causadas, por exemplo, por falta de atenção do desenvolvedor.

A coleta de lixo se tornou parte integral das linguagens de programação como Lisp, Smalltalk, Haskell, ML, Scheme, entre outras, no início dos anos 60, levando em consideração seus benefícios como confiança, dissociação do gerenciamento de memória das classes de interface e redução no tempo de desenvolvimento gasto pelos programadores para tratar o gerenciamento de memória [GOETZ *et al*., 2003].

Idealmente o coletor de lixo deve ser concebido como um agente cujas ações são transparentes, sendo executado em paralelo com a execução das aplicações. Isso implica que o coletor não causará interrupções na execução da aplicação em virtude da realização de uma coleta. Indo além, é também desejável que o coletor não cause impactos no desempenho da aplicação em função de sua interação com a memória virtual ou até mesmo com a *cache* [GOETZ, 2003].

Entretanto, por mais elaborado que seja o código de um coletor, este ainda necessita realizar processamento e, dependendo do algoritmo do ambiente de execução empregados, tal processamento pode causar ou mesmo exigir pausas na execução da aplicação, conseqüentemente

impactando negativamente o seu desempenho. Logo, percebe-se a importância do estudo do assunto objetivando o desenvolvimento de técnicas que reduzam os custos associados a coleta de lixo.

Antes de iniciar o estudo mais aprofundado das técnicas de coleta de lixo, é necessário apresentar a JVM (*Java Virtual Machine*), plataforma cujas técnicas de coleta de lixo serão estudadas, assim como outras definições que serão importantes no decorrer deste trabalho.

2.1 JVM (*Java Virtual Machine*)

A Java Virtual Machine é uma máquina virtual que funciona como uma plataforma para a execução de programas escritos na linguagem Java. Ela foi inicialmente desenvolvida visando uma solução para dispositivos consumidores de serviços da rede [LINDHOLM e YELLIN, 1999].

As aplicações desenvolvidas em Java são consideradas multi-plataformas, ou seja, em qualquer ambiente onde se tenha uma máquina virtual Java instalada, poderão ser executadas. Para que isso seja possível, os códigos-fonte são traduzidos para *bytecodes* que são interpretados ou convertidos em linguagem de máquina pela JVM.

De acordo com [Lindholm e Yellin, 1999], todas as implementações da JVM devem fornecer um coletor de lixo para reciclagem de memória (objetos não mais utilizados). Entretanto, o comportamento e a eficiência dos mesmos influenciam de forma direta no desempenho das aplicações que executam sob as mesmas [PRINTEZIS, 2005].

2.2 Objetos

O conceito de *objeto* surgiu com o paradigma de orientação a objetos e se refere à forma como são tratadas as estruturas de dados que fazem uso de tipos primitivos, referências a outros objetos e de métodos de tratamento desses tipos, todos encapsulados em uma classe, que, instanciada, se torna um objeto, ocupando espaço na memória.

A Figura 2.1 ilustra o processo de instanciação de um objeto em Java. A primeira linha representa o nascimento de um objeto, o que significa que nesse momento é alocado, na memória, um espaço para armazenamento de seus dados. As duas linhas seguintes representam

uma das formas de preencher seus atributos com tipos primitivos, que serão guardados na memória.

```
Empresa empresa = new Empresa();
empresa.setCod(3152);
empresa.setDescricao("Empresa Fictícia Ltda.");
```

Figura 2.1 Instanciação de Objetos em Java

Um objeto é considerado em uso quando há, pelo menos, uma referência a ele por outro objeto que também é referenciado. O ciclo de vida de um objeto varia, mas algumas situações são recorrentes, por exemplo: um objeto foi criado no escopo de um método que terminou sua execução e não o retornou, ou não fez com que o mesmo fosse referenciado por outro. Nesse momento, o objeto ainda permanece na memória, porém não há referências apontando para ele, ou seja, ele passa a ser considerado lixo.

Uma referência cíclica é criada quando existe um ciclo na cadeia de referências entre os objeto. Desta forma, partindo de um objeto inicial arbitrário e percorrendo suas referências, chegamos novamente ao objeto inicial. Estruturas como essa criam dificuldades para alguns algoritmos de coleta de lixo, como apresentaremos adiante.

2.3 *Heap*

A *heap* é um trecho de memória utilizado pelos processos para a alocação dinâmica de seus blocos de dados. Os objetos presentes na *heap* podem ser acessados de forma aleatória a qualquer momento. Geralmente refere-se à *heap* como um *array* contínuo de *slots* (blocos) onde os objetos serão mantidos. A Figura 2.2 apresenta um esquema simples de organização da *heap* para uma quantidade arbitrária de dados.

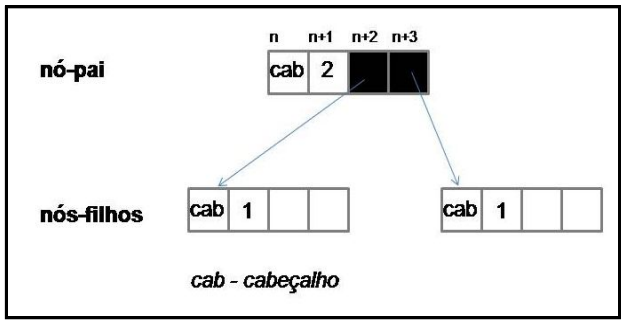

Figura 2.2 Blocos na heap – Adaptado de JONES e LINS, 1999

Na *Java Virtual Machine* pode-se definir, no momento de sua criação, o tamanho da *heap* que será alocada para a execução das aplicações.

2.4 Estratégias de Coleta de Lixo

É função do coletor de lixo realizar a coleta quando não há mais espaço para armazenamento dos novos objetos. A estratégia aplicada por cada algoritmo visa atender às demandas de cada perfil de aplicação e diferem na forma como realizam a limpeza.

Entretanto, a diversidade de estratégias apresenta certos padrões e uma uniformidade nos mecanismos de detecção de objetos não mais utilizados. Para se desenvolver um algoritmo de busca por objetos não mais utilizados, em geral, depara-se com a dificuldade em determinar quais objetos não são mais alcançáveis por um aplicativo em execução. A partir daqui, serão considerados os objetos não mais alcançáveis como objetos lixo e os alcançáveis como vivos.

Segundo Goetz, os blocos de memória podem ser alcançáveis de duas maneiras: o aplicativo mantém uma referência a um bloco por uma variável estática (*static* – instanciado estaticamente e que tem seu ciclo de vida durante toda a execução da aplicação) ou quando há referência ao bloco por outro bloco que também é alcançável.

Ao longo dos anos foram desenvolvidas diversas técnicas de coleta de lixo para atender às demandas de cada perfil de aplicação, sempre visando a transparência e a redução no custo da coleta. Algumas delas serão abordadas nos próximos tópicos.

Duas das principais abordagens de técnicas para coleta de lixo são a *Reference Counting* (Contagem por Referência) e *Tracing* (Rastreamento) [VENNERS, 2009].

2.4.1 *Reference Counting* (Contagem por Referência)

Reference Counting é uma das técnicas mais antigas de coleta de lixo. As estratégias baseadas em Contagem por Referência distinguem os objetos vivos dos lixos mantendo um contador para cada objeto presente na *heap*. O contador contém, assim, o número de ponteiros que apontam para o objeto presente em um dado bloco [VENNERS, 1996].

Técnicas de contagem por referência necessitam do auxílio do compilador, visto que, durante a execução do aplicativo, sempre que uma nova instância de uma classe for criada, é necessário incrementar o contador; assim como no momento de destruição do objeto, é necessário decrementá-lo. Tais operações só são possíveis caso haja trechos de código capazes de incrementar, decrementar e manter os contadores.

Um objeto é considerado lixo quando seu contador tem o valor 0 (zero), indicando que nenhum bloco da *heap* mantém referência para seu bloco, ou seja, a aplicação não faz mais uso daquele objeto e seu espaço em memória pode ser utilizado por outro objeto.

O que caracteriza os algoritmos de contagem por referência é que eles possuem baixo custo de execução e realizam operações simples, porém, a eficiência dessas operações é afetada em aplicações onde o número de referências é grande e vários objetos são criados, visto que o custo de atualização dos contadores é alto [MORAES, 2007].

Uma outra limitação desta técnica está relacionada à detecção de referências cíclicas. Suponha que sejam instanciados três objetos, A, B e C. O objeto A faz referência a B. B, por sua vez, faz referência a C e C referencia A. Suponha finalmente que A seja referenciado por apenas um objeto, por exemplo, D, e que os demais objetos, B e C, não tenham nenhuma outra referência além das já citadas. Neste cenário, o contador de A é igual a 2, enquanto os contadores de B e C são iguais a 1. Caso D não faça mais referência a A, o contador de A será 1, porém não existe nenhuma outra referência externa que mantenha A e os objetos que possam ser acessados a partir dele vivos. A Figura 2.3 ilustra esta situação.

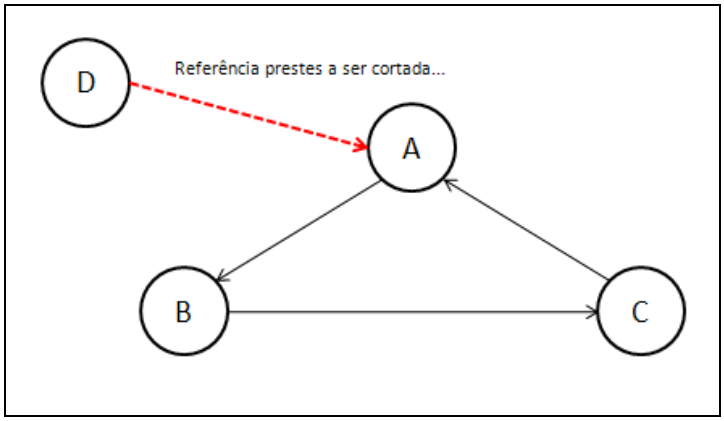

Figura 2.3 Referência cíclica não detectada por técnicas de contagem por referência

Em virtude dessas limitações, as máquinas virtuais Java optam por utilizar algoritmos de rastreamento para gerenciamento do lixo na *heap*. Entretanto, segundo [VENNERS, 2009], essa limitação não impede que os algoritmos de contagem por referência sejam utilizados. Por exemplo, sistemas *real-time* (tempo real) utilizam a contagem por referência pela previsibilidade do seu tempo de execução, característica de suma importância para esta classe de sistemas.

2.4.2 *Tracing* (Rastreamento)

Algoritmos de rastreamento para coleta de lixo consideram as referências entre objetos como um grafo que se inicia do nó-raiz (*root*). A partir dessa estrutura é possível percorrer todas as referências através de heurísticas de busca em árvores. Deve-se levar em consideração, também, a transitividade entre os nós, ou seja, se um nó A alcança um nó B e B alcança um nó C, então A alcança C.

O nó-raiz é composto por "*variáveis do programa do usuário, parâmetros da função em execução, variáveis globais, dentre outros*". Quando se alcança um nó durante a busca, significa que há um ponteiro apontando para ele e a partir disso é possível concluir que o objeto presente no bloco ainda está sendo utilizado pela aplicação, ou seja, não deve ser afetado pela limpeza na memória [MORAES, 2007].

Uma das principais características dos algoritmos de rastreamento é a perda de desempenho imposta à aplicação. Isso acontece pelo fato de ocorrerem pausas no processo de rastreamento para que o coletor percorra todos os objetos alcançáveis na *heap* a partir do nó-raiz.

A perda de desempenho causada não é um empecilho à sua utilização na Java Virtual

Machine. De fato, a maioria dos coletores presentes na mesma faz uso de heurísticas contendo estratégias de rastreamento, sendo o *mark-sweep* o mais conhecido de todos.

a) Mark-sweep

O coletor *Mark-sweep* é a forma mais básica de um algoritmo de rastreamento. Foi proposto por John McCarthy em 1960 e introduzido na linguagem Lisp. Quando é executado, a aplicação pára e seu coletor visita cada nó (objeto vivo) partindo do nó-raiz, marcando cada um deles. Quando não há mais referências, é realizada a reciclagem, ou varredura (*sweep*), dos objetos não marcados e que representam o lixo [GOETZ, 2009]. A Figura 2.4 ilustra uma situação contendo blocos alcançáveis e não alcançáveis (sombreados).

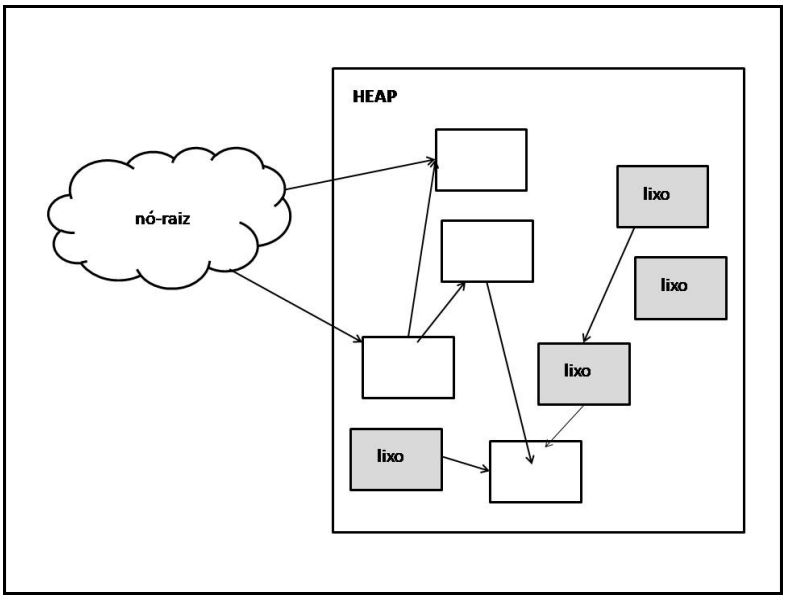

Figura 2.4 Objetos alcançáveis e lixos – Adaptado de [GOETZ, 2009]

É possível, assim, desmembrar a execução do *mark-sweep* em duas fases: marcação e varredura. Nessa primeira fase, o algoritmo percorre a *heap* marcando (*mark*) todos os objetos alcançáveis a partir do nó-raiz. Após a leitura e marcação, é iniciado o processo de varredura, onde todas as referências são consultadas e os objetos não marcados são reciclados, o que é chamado de troca (*sweep*).

Sendo um dos primeiros algoritmos de coleta de lixo criados, é comum encontrar inconvenientes em sua utilização. A fase de varredura dos objetos marcados gera uma pausa na aplicação que é proporcional ao tempo que o coletor precisa para realizá-la, ou seja, quanto maior a memória utilizada, maior o tempo de pausa [MORAES, 2007].

A fragmentação é outro ponto negativo desta técnica. Ela é decorrente da liberação dos espaços de memória não mais utilizados sem que seja realizada uma reorganização do conteúdo restante da memória. Esta situação é ilustrada na Figura 2.5.

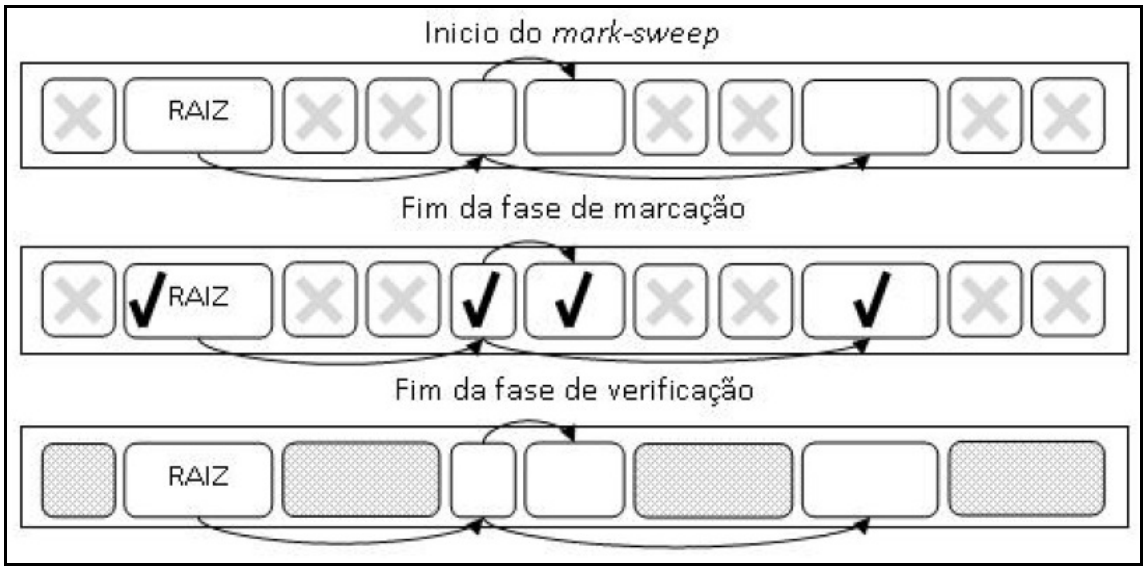

Figura 2.5 Algoritmo mark-sweep [MORAES, 2007]

Na tentativa de evitar a fragmentação das regiões de memória, algumas estratégias foram desenvolvidas, como por exemplo o *mark-compact collector*. Este coletor é um híbrido entre o coletor *mark-sweep* e o *copying collector*. Ele utiliza a marcação e varredura do *mark-sweep* em conjunto com o *copying collector*, que realiza uma realocação dos objetos ainda alcançáveis na memória, utilizando para isso segunda região de memória. O resultado final é um espaço de memória sem fragmentação.

b) Copying Collector

Para utilização da técnica de *copying collector* (coleta por cópia) a *heap* é dividida em duas regiões de mesmo tamanho. Durante a execução da aplicação, apenas uma das regiões é

utilizada (região ativa). Quando há necessidade de alocação de memória, ou seja, não há mais blocos disponíveis na região ativa, o algoritmo pára a aplicação, realiza uma busca por objetos alcançáveis e em seguida os copia para o espaço inativo. Terminada a cópia, o espaço inativo se torna ativo e vice-versa.

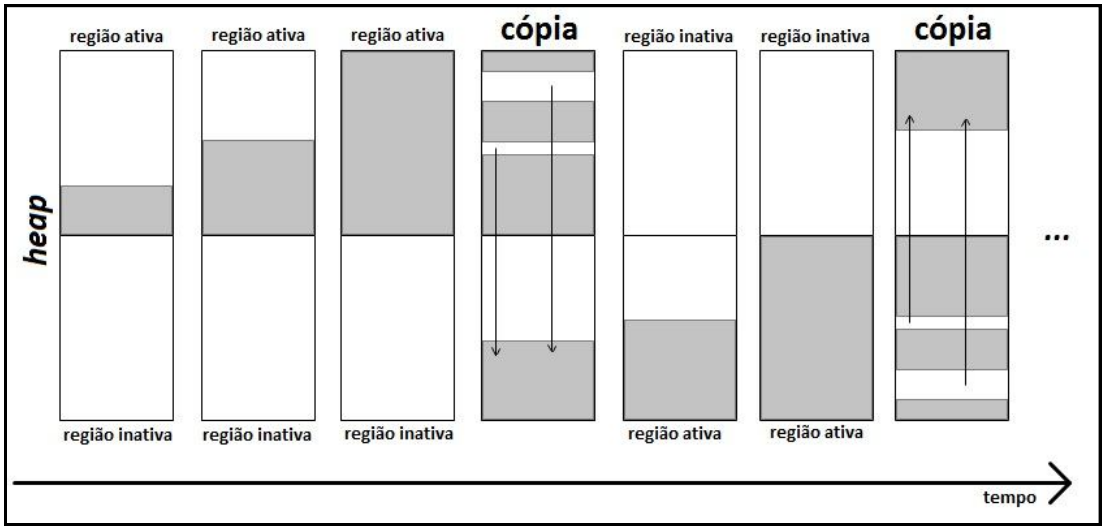

Figura 2.6 Processo de troca entre regiões – Adaptado de VENNERS, 1996

A duração do processo de cópia é diretamente proporcional ao número de objetos vivos, sendo que, para cada objeto copiado é necessário, também, atualizar as referências ao mesmo para a nova região de memória, o que representa um custo a mais para sua execução.

Seu uso é recomendável para cenários onde o número de objetos considerado lixo é grande – essa situação é obtida quando há uma grande quantidade de alocação de objetos com ciclo de vida curto. Para cenários onde o ciclo de vida dos objetos é longo, seu uso não é recomendável, visto que os mesmos objetos serão copiados inúmeras vezes.

2.5 Coleta de Lixo na Java Virtual Machine

O número de coletores presente na Java Virtual Machine aumentou das primeiras versões à atual. Na JDK 1.3 havia apenas 3 (três) estratégias de coleta de lixo, ao passo que a versão 1.4.1 fornece 6 (seis) e mais uma série de comandos para que seja possível configurá-las [GOETZ, 2003].

Um aspecto importante dos coletores da Máquina Virtual Java é que as suas distintas

implementações visam distintos perfis de aplicação. Assim, o usuário pode escolher o coletor mais adequado as características de sua aplicação [GOETZ, 2003].

Os coletores presentes na JVM são considerados exatos, isto é, garantem que todos os objetos não mais utilizados serão coletados e permitem que todos os objetos sejam realocados, eliminando a fragmentação da memória e aumentando a localidade [MORAES, 2007].

Uma técnica utilizada pelos coletores presentes na JVM é a chamada coleta por geração (*generational collection*), onde a *heap* é dividida em regiões (gerações) que levam em conta a idade do objeto, sendo que, para cada região, é utilizada uma técnica diferente para coleta [MORAES, 2007].

O conceito de geração se baseia no tempo de vida de um objeto, que pode ser curto, longo ou permanente. Para isso são utilizadas as gerações jovem, antiga e a permanente, respectivamente. Objetos recém-criados fazem parte da geração jovem e podem ser promovidos às gerações antiga e permanente, de acordo com heurísticas específicas de controle de geração.

2.5.1 Geração Jovem

A geração jovem é um espaço na memória dedicado aos objetos recém-criados e de tempo de vida curto, resultando em uma região de memória onde constantemente há alocação de blocos. Por essa razão, o número de coletas na geração jovem é maior que nas outras gerações, visto que quanto mais utilizada, maiores são as chances de completarem sua capacidade de armazenamento.

Dentro deste espaço, há outra divisão que classifica ainda mais seus objetos: o Éden e o espaço dos sobreviventes de origem e destino. Objetos recém-criados vão para o Éden e podem ser promovidos ao espaço dos sobreviventes.

É no Éden que é alocada a maioria dos objetos criados, visto que, a princípio, praticamente todos os objetos terão vida curta – salvo exceções, onde a alocação de um objeto considerado grande utiliza diretamente as gerações antiga e permanente.

A coleta de lixo realizada no Éden tem um papel fundamental na manutenção das regiões da geração jovem. Quando não há mais espaço para alocação de novos objetos, uma coleta de lixo fará com que haja somente objetos sobreviventes nessa região. Partindo desta observação, a JVM trabalha com a hipótese de que quanto mais coletas um objeto sobrevive, maiores as chances desse objeto viver por ainda mais tempo [MORAES, 2007] . Assim, os objetos do Éden

que sobrevivem a um número de coletas de lixo definido na especificação da JVM, são movidos para o espaço dos sobreviventes.

O *copying collector* é o algoritmo utilizado no espaço dos sobreviventes. Para que ele funcione, este espaço é dividido em espaço de origem e espaço de destino.

Os objetos que vêm do Éden são alocados na região de origem, e quando esgota-se o espaço disponível, é realizada uma coleta que irá enviar os objetos vivos para a região de destino. No final do processo os nomes dos espaços mudam: a região de destino passa a ser chamada de origem e vice-versa. Um outro espaço existente é a chamada região antiga, que é para onde os objetos que não cabem no espaço de sobreviventes são enviados, bem como os objetos que constantemente sobrevivem às coletas.

O fluxograma presente na Figura 2.7 ilustra o caminho percorrido por um objeto ao longo de seu ciclo de vida nas gerações jovem, antiga e permanente. O processo que utiliza o algoritmo do *copying collector* entre as regiões de origem e destino possuem algumas características semelhantes ao ilustrado na figura 2.6.

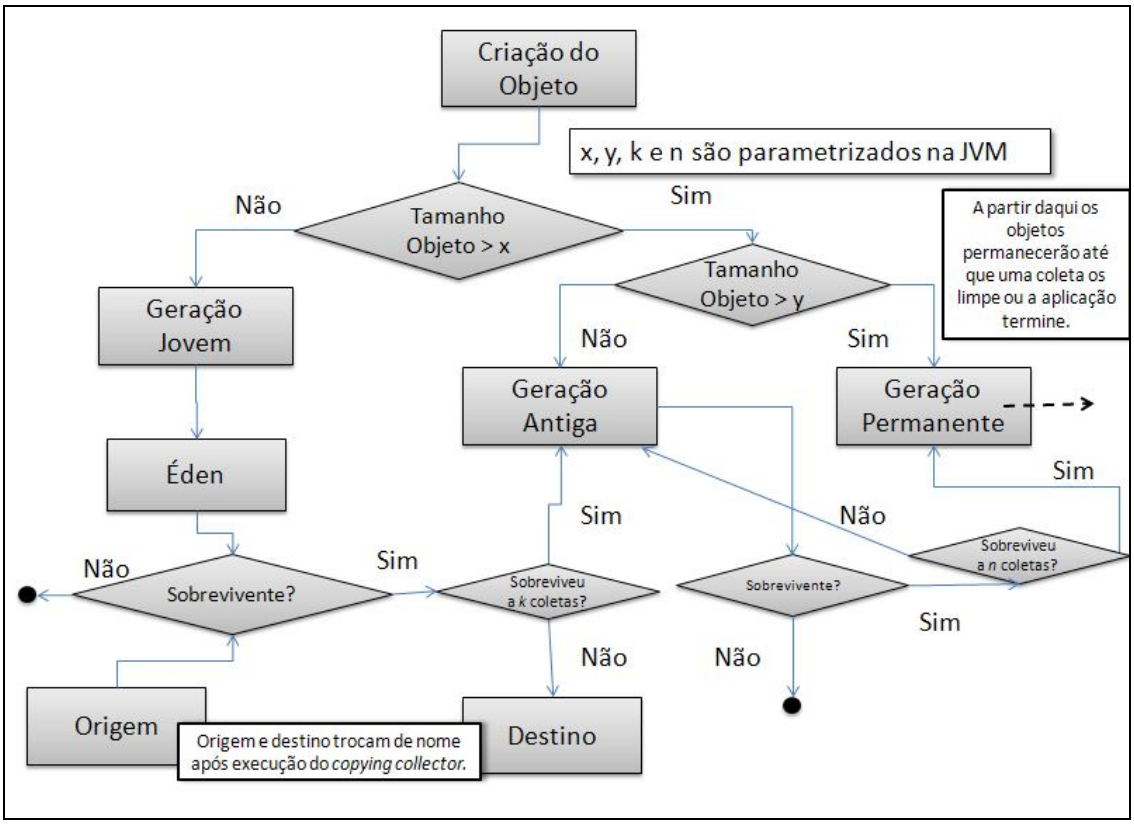

Figura 2.7 Promoção e destruição de objetos nas regiões da Geração Jovem

2.5.2 Geração Antiga

Os objetos sobreviventes ou que foram promovidos ainda na criação como descrito na seção anterior fazem parte de uma região, considerada “amadurecida”, chamada de geração antiga. Estes são os possíveis objetos de vida mais longa e que podem se tornar objetos de tempo de vida permanente. Essa região possui um espaço contíguo na memória possuindo tamanho maior que a geração jovem [MORAES, 2007].

Em conseqüência disso, as coletas efetuadas na geração antiga têm proporções maiores e que merecem um tratamento mais inteligente. Por exemplo, se houver necessidade de realizar uma coleta na geração jovem, mas esta contiver apenas objetos alcançáveis, é indispensável que haja uma coleta na geração antiga a fim de promover os objetos jovens, caracterizando uma situação onde a coleta na geração antiga foi invocada por outra geração, mesmo havendo espaço para novos objetos.

2.5.3 Coletores na JVM

A JVM fornece três tipos de coletor para a geração jovem: em série, em paralelo e o concorrente; sendo o primeiro deles o utilizado por padrão. De qualquer forma, o coletor pode variar de acordo com as características da aplicação [MORAES, 2007].

Ainda segundo Moraes, o *mark-compact* é o algoritmo responsável pela coleta em série. Suas principais características são a de ser implementado para ambientes mono-processados e ser do tipo “pare o mundo”. Já o algoritmo de coleta em paralelo utilizado é o *mark-sweep* em conjunto com o *copying collector* – união também observada no espaço de sobreviventes da geração jovem – onde há um processo de coleta dividido em várias *threads*. E por fim, a coleta concorrente é feita pelo *concurrent mark-sweep,* implantado na JVM com o objetivo de reduzir o tempo de latência em aplicações com tamanho de *heap* elevado, tendo em vista que as coletas na geração antiga são as principais responsáveis pelas longas pausas causadas por coletores.

Na busca por uma melhor eficiência no algoritmo *mark-sweep,* proporemos algumas mudanças em sua estrutura de forma a torná-lo “preguiçoso”, isto é, realizando uma coleta menor, mas que não influencie no resultado final para a aplicação.

Tendo apresentado os algoritmos presentes na JVM e como eles são utilizados em suas gerações, é possível, agora, apresentar os aperfeiçoamentos propostos.

3 *Mark-sweep*

O *mark-sweep*, também conhecido como *mark-scan*, foi o primeiro algoritmo de automação para gerenciamento de memória e faz parte do grupo dos coletores de rastreamento [MCCARTHY, 1960]. A principal vantagem deste algoritmo sobre os coletores com contagem de referência é o fato de não haver necessidade de ações especiais para que haja reciclagem dos objetos [JONES e LINS, 1999].

Neste algoritmo, os objetos não são reciclados no momento em que se tornam lixo [JONES e LINS, 1999]. Inicialmente, a aplicação solicita espaço na *heap* para armazenar novos objetos. Caso a *heap* esteja cheia, ou não exista espaço livre para a alocação dos objetos devido à fragmentação, é realizada uma chamada ao coletor que iniciará um processo de detecção de objetos vivos para, então, reciclar os mortos.

Como dito anteriormente, seu funcionamento é simples e se resume a duas rotinas básicas, sendo a primeira delas de marcação de objetos lixo – que envolve uma busca na *heap* – e a segunda o processo de reciclagem dos objetos não-marcados.

A figura 3.1 apresenta os pseudo-códigos das duas fases do *mark-sweep* dividas em três rotinas.

```
mark_sweep() =
    for R in Roots
        mark(R)
    sweep()
    if free_pool is empty
        abort "Memória Cheia"

mark(N) =
    if mark_bit(N) == unmarked
        mark_bit(N) = marked
        for M in Children(N)
            mark(*M)

sweep() =
    N = Heap_Bottom
    while N < Heap_top
        if mark_bit(N) == unmarked
            free(N)
        else mark_bit(N) = unmarked
        N = N + size(N)
```

Figura 3.1 O coletor mark-sweep – Adaptado de [JONES e LINS, 1999]

É necessário destacar que não há, no código da figura anterior, uma definição formal do mecanismo a ser adotado para a liberação da memória, visto que este é um aspecto característico da implementação do algoritmo e que, portanto, pode variar entre as suas diversas formas de implementação. Da mesma forma, não foi definido formalmente o que ocorrerá nas situações onde não exista memória disponível após o processo de coleta. Uma das possíveis soluções para evitar abortar a execução da aplicação é aumentar o tamanho da *heap* após um processo de coleta que não resulte em espaços livres. Entretanto, tais variações fogem ao escopo do *mark-sweep*.

É possível indicar o estado (vivo ou morto) de um objeto através de um *bit* que é habilitado quando o objeto é alcançado. Este *bit* será consultado no momento em que o algoritmo percorre linearmente a *heap*: objetos onde o *bit* de marcação está desmarcado serão reciclados [JONES e LINS, 1999].

Resumindo, quando uma aplicação solicita uma certa quantidade de memória para alocação de um novo objeto, mas não há espaço na *heap*, o coletor é chamado. No caso do *mark-sweep*, inicia-se a marcação pelo nó raiz, e visitando-se então todos os objetos alcançáveis a partir dele. A partir daí, cada objeto alcançável se torna a raiz da próxima busca, de forma que são alcançados todos os objetos que ainda estão sendo utilizados pela aplicação. Quando um objeto é alcançado, seu *bit* de marcação é habilitado. Terminada a fase de marcação, a *heap* é percorrida linearmente, e todos os objetos que possuam o *bit* de marcação desabilitado são reciclados, sendo possível alocar novos objetos nestas posições de memória.

A JVM possui um coletor que implementa uma variação do algoritmo *mark-sweep* básico aqui apresentado, chamado *mark-compact*. Ele é dividido em quatro fases: a) marcação e reciclagem, b) computação dos novos endereços de memória dos objetos vivos, c) ajuste dos ponteiros e d) compactação. Tais fases são necessárias já que o algoritmo básico pode levar a uma grande fragmentação de memória.

3.1 Desempenho

O desempenho dos algoritmos de coleta de lixo é um dos fatores determinantes para a escolha do método mais adequado a ser utilizado em um ambiente que implemente o gerenciamento automático de memória.

Dentre os fatores que justificam o uso do *mark-sweep* sobre os algoritmos de contagem por referência pode-se citar o melhor manuseiamento de estruturas que podem resultar em ciclos no percurso dos objetos [JONES e LINS, 1999].

A principal desvantagem do uso do *mark-sweep* está no fato deste ser um algoritmo do tipo “pare o mundo”, o que significa que a aplicação deve esperar pelo coletor para continuar sua execução. O tempo de pausa é proporcional ao tamanho da *heap*, visto que todos os objetos vivos serão visitados, assim como toda a *heap* será percorrida após a fase de marcação. Desta forma, não podemos desconsiderar o impacto negativo no desempenho da aplicação que o uso deste algoritmo pode causar [JONES e LINS, 1999].

Por este ser um aspecto de grande importância, no início dos anos 80, Fateman em Caracterization of VAX Macsyma, 1981, observou que programas em Lisp gastam entre 25-40% do tempo realizando as fases de marcação (mark) e troca (sweep), sendo o tempo de espera do usuário de, aproximadamente, 4,5 a cada 79 segundos [JONES e LINS, 1999].

Tais resultados levaram estudiosos da área, como Fateman, a constatar que a utilização de algoritmos *mark-sweep* não é recomendada para sistemas de tempo real, de alta interatividade ou até mesmo para sistemas distribuídos. Grandes pausas representam perdas irreparáveis em sistemas com essas características.

3.2 Fragmentação

O algoritmo *mark-sweep* básico tende a fragmentar a memória após um certo número de execuções pelo simples fato de focar apenas da reciclagem dos objetos mortos; ou seja, terminada a execução do algoritmo, os locais onde eles se localizavam se tornam regiões livres onde novos objetos poderão ser alocados. Após realizar várias coletas, a quantidade de memória livre pode até ser maior do que se tinha no início do processo, mas esta estará mais distribuída. Em determinadas situações, a memória pode estar tão fragmentada que não existam espaços contíguos suficientes mesmo para a alocação de pequenos objetos na *heap*.

A figura 3.2 exemplifica o processo de fragmentação da *heap* a medida que são executadas, por exemplo, três coletas. Será ignorado, nesta figura, o fato de que o *mark-sweep* é chamado apenas quando não há mais endereços livres, visto que um objeto que requer um espaço grande e contíguo na *heap* também pode disparar uma chamada ao coletor.

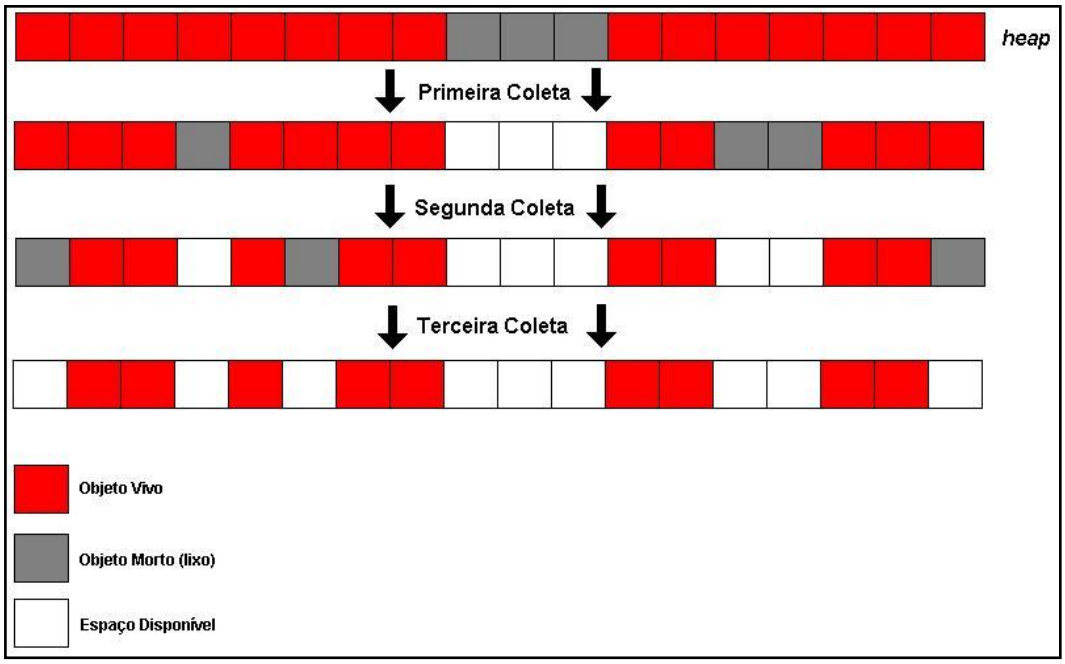

Figura 3.2 Processo de fragmentação após coletas com *mark-sweep*

O processo de fragmentação na JVM é tratado realizando-se uma compactação nas gerações que utilizam o *mark-sweep*, conforme destacado no código da figura 3.3. Este código foi retirado do arquivo **genMarkSweep.cpp**, mais especificamente do método responsável pela quarta fase do algoritmo de coleta.

```
GenCollectedHeap* gch = GenCollectedHeap::heap();
Generation* pg = gch->perm_gen();

EventMark m("4 compact heap");
TraceTime tm("phase 4", PrintGC && Verbose, true, gclog_or_tty);
trace("4");

VALIDATE_MARK_SWEEP_ONLY(reset_live_oop_tracking(true));

pg->compact();

VALIDATE_MARK_SWEEP_ONLY(reset_live_oop_tracking(false));
```

Figura 3.3 Chamada para compactação da Geração no mark-sweep pela JVM

3.3 Espaço e Localização

Em determinadas situações, os objetos considerados lixo são salvos pelo coletor em uma lista ou vetor. Ao utilizar um ponteiro para cada um desses objetos e liga-lo à lista pode-se gerar

cache miss – erro de cache, que ocorre quando um aplicativo solicita à cache algum dado que ela não possui – ou falta de página – que é uma interrupção disparada pelo hardware quando tenta-se acessar uma página na memória que não foi carregada no espaço físico do sistema [JONES e LINS, 1999].

3.4 *Stack overflow*

Stack (pilha) – ou *call stack* – é uma estrutura de dados dinâmica que armazena informações a respeito das sub-rotinas de uma determinada aplicação em execução na máquina. Nesse contexto, cada linha de código do programa está armazenada em um índice no *call stack*, que, quando terminada deve retornar à linha que realizou a chamada para o procedimento que estava ativo.

Quando a quantidade de memória utilizada pelos programas no *call stack* atinge um limite máximo, tem-se o *stack overflow*, que nada mais é do que um estouro de pilha, ou seja, não há mais espaço para armazenamento de memória e o programa termina de forma inesperada e incorreta.

Uma das principais dificuldades na fase de marcação de objetos no *mark-sweep* é que ela é executada no momento em que a aplicação necessita de memória, ou seja, o coletor acaba trabalhando com o mínimo de recursos do sistema [JONES e LINS, 1999].

Nesse cenário, é possível introduzir o *marking stack* – pilha de marcação utilizada pelo *mark-sweep* para evitar problemas com *overflow*. Parte-se do princípio que com a utilização deste mecanismo é possível prever estouros e tomar as medidas cabíveis antes que este aconteça.

Durante a execução do *mark-sweep*, operações de *push* e *pop* – inserção no topo e retirada do topo, respectivamente – são realizadas no *marking stack* enquanto os objetos são alcançados e marcados.

Jones e Lins citam duas formas de se detectar *overflow* com *marking stacks*. A primeira – e mais simples – delas, é realizar uma checagem toda vez que houver uma operação de *push* (inserção no topo da pilha). A outra forma – e mais eficiente – é contar o número de ponteiros que partem do nó retirado do topo da pilha, ou seja, toda vez que for realizado um *pop*.

A JVM implementa o *marking stack* como sendo do tipo *GrowableArray* – estrutura em forma de *array* com tamanho dinâmico; isto é, pode crescer ou diminuir durante a execução da aplicação sem que haja necessidade de tamanho fixo definido. A figura 3.4 foi retirada do código-fonte do arquivo **markSweep.cpp** e apresenta a invocação, por parte da JVM, da operação de *marking stack*.

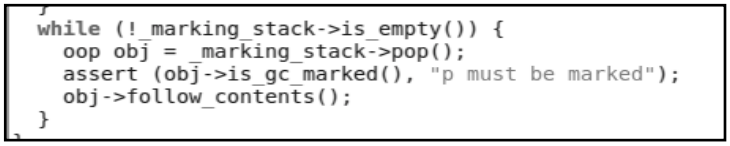

Figura 3.4 Exemplo de utilização do *marking stack* pela JVM

A estrutura acima está presente em um método chamado durante a fase de marcação dos objetos. Enquanto o *marking stack* contém objetos (que neste contexto estão vivos), retira-se o objeto presente no topo da pilha através de `marking_stack->pop();` e em seguida faz uma checagem para garantir que o objeto está marcado. Por fim, realiza-se uma chamada a `follow_contents()` que irá buscar todos os objetos alcançáveis partindo-se do objeto retirado do topo da pilha (*marking stack*) – que passa a ser a raiz (*root*).

Em `follow_contents` os objetos alcançáveis são marcados e adicionados ao topo da pilha, completando o ciclo de recursão, como mostra a figura 3.5. Os objetos no topo da pilha também sofrerão chamadas para alcançar outros objetos e assim por diante.

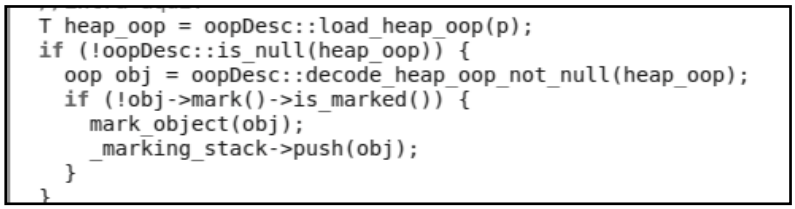

Figura 3.5 Marcação e adição de objetos ao topo do *marking stack* na JVM

O trecho acima foi retirado do arquivo **markSweep.inline.hpp** e é responsável por marcar o objeto – visto que ele foi alcançado – em `mark_object` e em seguida adicionar o

mesmo ao *marking stack.*

As rotinas apresentadas neste tópico compõem praticamente a primeira fase principal do *mark-sweep*, que é a fase de marcação. Ao final dela, os objetos na *heap* que ainda estão sendo utilizados pela aplicação possuem o *bit* de marcação com o valor que o representa como vivo. A forma como a marcação dos *bits* é feita está presente no próximo tópico.

3.5 *Bits* de Cabeçalho e Mapas de *bits*

Como visto no início do capítulo, para armazenar o estado (vivo ou morto) de um objeto na *heap*, utiliza-se, geralmente, um *bit* de marcação associado a cada objeto. Muitos sistemas armazenam o *bit* no próprio cabeçalho como um inteiro de tamanho pequeno (*small integer*) [JONES e LINS, 1999]. Outros sistemas armazenam o *bit* no endereço referente ao objeto na memória [FODERARO, *et al*, 1985] ou até mesmo em ponteiros do objeto [STEENKISTE, 1987].

Uma maneira efetiva de armazenar informações do estado do objeto é a utilização de um mapa de *bits*. Dessa forma, não se perde espaço no cabeçalho nem em outras estruturas [JONES e LINS, 1999].

Para implementação do mapa de *bits*, pode ser utilizada uma lista linear (um simples *array*, por exemplo), visto que o espaço ocupado pela estrutura na *heap* é inversamente proporcional ao tamanho do menor objeto alocável na mesma.

A vantagem de se utilizar mapas de *bits* está na centralização das informações a respeito do estado dos objetos, que, quando gravadas no cabeçalho, se tornam dispersas ao longo da *heap*. Além disso, a limpeza (gravação de zeros, por exemplo) dos *bits* de marcação é feita com facilidade quando estes estão próximos uns dos outros [GARNER, *et al*., 2007].

A JVM implementa dois mapas de marcação como forma de obter um sumário dos objetos vivos presentes na *heap*. Nestes mapas, cada *bit* corresponde a uma palavra de 64 *bits*. Um dos mapas é utilizado para a coleta ativa e o outro para a coleta previamente efetuada [BRUNO, 2009]. A figura 3.6 representa um esquema de como a JVM mantém um mapa de *bits*.

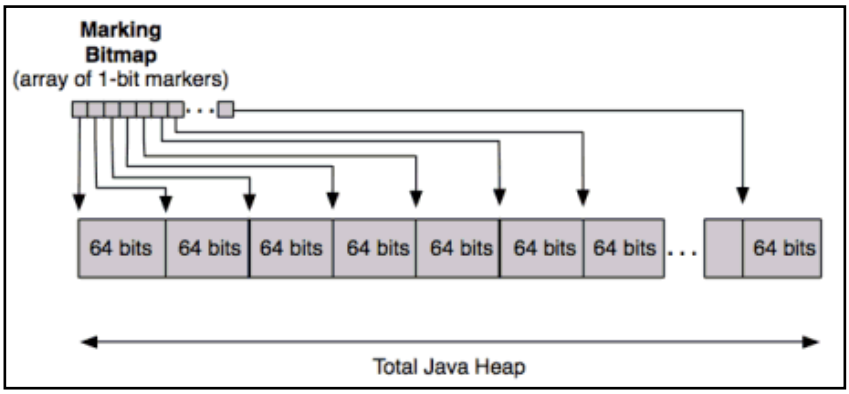

Figura 3.6 Marking Bitmap (mapa de marcação de bits) na JVM [BRUNO, 2009]

3.6 *Sweep*

Os coletores *mark-sweep* executam a varredura – processo responsável por reciclar objetos mortos – da *heap* após a fase de marcação, identificando objetos não-marcados e preenchendo as listas de blocos livres [GARNER, *et al*., 2007].

[GARNER, *et al*., 2007] destaca ainda que a fase de varredura pode ser desmembrada em duas rotinas principais. A primeira delas se refere ao exame dos blocos de metadados para identificação dos objetos marcados. A segunda rotina preenche as listas de blocos livres (ou o mecanismo utilizado pela linguagem para manter endereços de blocos livres na memória) com os locais alocados pelo coletor.

É nesse contexto que se destacam os mapas de *bits*, descritos anteriormente. O custo de se criar uma lista para blocos livres é alto, visto que operações em uma lista geralmente envolvem iterações por cada bloco. Já com os mapas de *bits*, solicitações de blocos livres, assim como de alocações, são feitas de modo mais simples e eficiente.

É possível perceber que o processo de alocação de memória utilizando a varredura do *mark-sweep* é um processo custoso. Garner caracteriza-o como *eager sweeping*, ou varredura gulosa. A idéia de otimizar o processo de realocação deu origem ao *lazy sweeping*, ou varredura preguiçosa; e que será detalhado mais adiante.

3.7 *Lazy Sweeping*

Algoritmos *mark-sweep* são do tipo “pare o mundo”. Isto é, a execução de um programa deve ser pausada para que o coletor realize a limpeza da memória – o que os impede de serem utilizados em sistemas de tempo real, por exemplo.

Devido à essa característica, diversas técnicas para melhorar seu desempenho foram criadas ao longo da história. [HUGHES, 1982] descreve uma destas técnicas: a dos coletores incrementais, onde há uma quantidade pré-definida de marcações para reduzir o tempo de pausas na primeira fase do algoritmo. Tal alteração gera o problema das mutações, que ocorre quando há alterações no grafo de objetos da *heap* interferindo no grafo de marcações do coletor.

O conceito de *lazy sweeping* é conhecido e utilizado há mais de uma década; introduzido como um mecanismo de redução de paginação e pausas [BOEHM, 2000]. Serão descritos, nos próximos tópicos, três distintas implementações propostas por Hughes [HUGHES, 1982], Boehm e Weiser [BOEHM e WEISER, 1988] e Zorn [ZORN, 1989]. Tais propostas foram analisadas por [JONES e LINS, 1999].

Lazy Sweeping por Hughes

Foi dito anteriormente que um dos motivos que levam a um maior custo computacional dos algoritmos *mark-sweep* incrementais é o tratamento das alterações efetuadas nos objetos presentes na *heap* após os mesmos terem sido tratados pelo coletor.

Durante a fase de varredura esse problema não acontece, já que as rotinas de leitura e varredura na *heap* são transparentes às aplicações. A maneira mais simples, então, de manter programa e coletor em paralelo é permitir apenas uma quantidade pré-determinada de varreduras a cada alocação.

A figura 3.7 representa o algoritmo de Hughes para *lazy sweeping*, sendo que, a cada vez que `allocate` é chamado, a varredura inicia e termina até que a quantidade de memória solicitada seja alcançada.

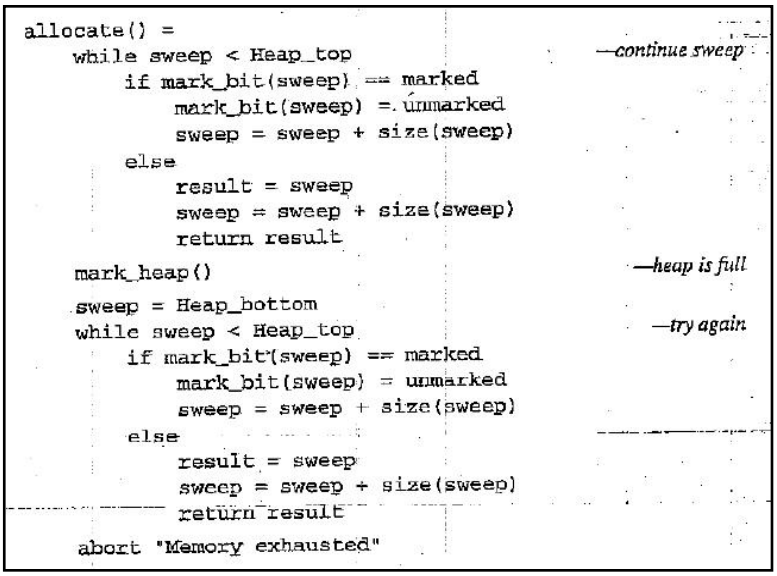

Figura 3.7 Lazy Sweeping por Hughes [JONES e LINS, 1999]

Uma das vantagens de se utilizar *lazy sweeping*, como mostrado acima, é que o custo da fase de varredura é transferido para a fase de alocação, que agora passa a ser controlado, não mais varrendo a *heap* por inteiro [HUGHES, 1982].

Entretanto, para sistemas que utilizam mapas de *bits*, esta estratégia não é a mais recomendável. Visto que tais estruturas necessitam ser atualizadas a cada coleta – sendo os mapas de *bits* uma ou mais palavras de *bits* – as chamadas a `allocate` terão que trabalhar com uma rotina de leitura/alteração dessas palavras, o que se torna, em determinados casos, custoso [JONES e LINS, 1999].

A varredura por Boehm e Weiser

A fase de alocação é executada em duas fases [BOEHM e WEISER, 1988]. Na primeira delas, um alocador adquire blocos de 4Kb (valor que pode variar) utilizando um alocador do sistema operacional – `malloc`, por exemplo. Para evitar fragmentação, cada bloco contém apenas objetos de tamanho único. A figura 3.8 contém uma representação da estrutura utilizada por Boehm e Weiser.

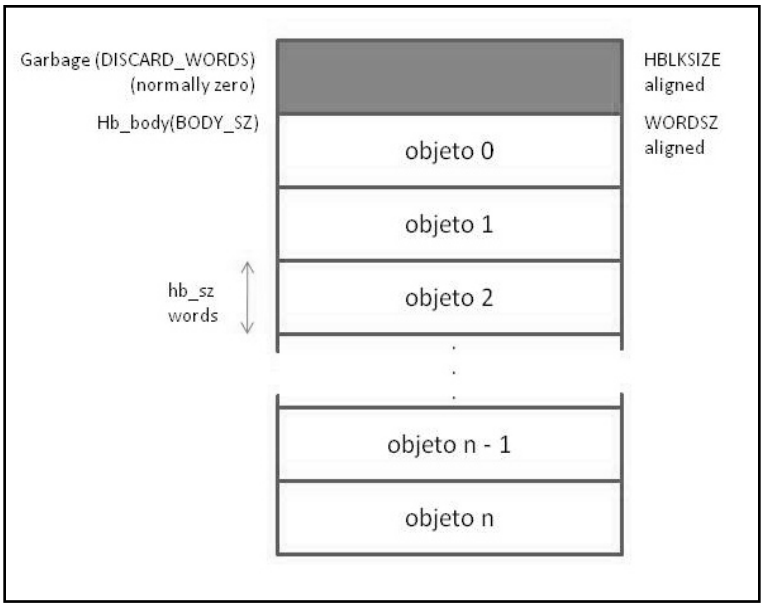

Figura 3.8 Estrutura de bloco na varredura de Boehm e Weiser [JONES e LINS, 1999]

Cada bloco na estrutura acima possui um cabeçalho contendo informações como: tamanho dos objetos presentes no bloco, próximo bloco, descritor de objeto para marcação, tipo de objeto, *bits* de marcação, entre outros.

Através da aquisição de novos blocos obtidos do sistema operacional, a *heap* pode ser expandida a qualquer momento – o que ocorre quando o coletor termina sem ter reciclado a quantidade de memória solicitada pelo programa.

Lazy Sweeping por Zorn

Zorn aborda de uma forma mais complexa a fase de varredura para implementar *lazy sweeping*. Ao invés de utilizar listas de blocos livres, aloca-se um vetor em forma de *cache* com *n* objetos para cada tamanho de objeto conhecido. Se o vetor solicitado está vazio, a *heap* é varrida para que seja populado. É uma estratégia que diminui bastante o tempo de pausa da fase de varredura tendo em vista que, cada iteração da varredura verifica apenas uma palavra no mapa de *bits*, inserindo os nós de blocos livres nos vetores [JONES e LINS, 1999].

A figura 3.9 contém o pseudo-algoritmo de Zorn para *lazy sweeping*.

```
— is a collection needed?
    subcc %g_allocated, ConsSize, %g_allocated
    bg,a noCollect
    subcc %g_freeConsIndex, 4, %g_free_ConsIndex
    call Collect
    nop
    subcc %g_freeConsIndex, 4, %g_free_ConsIndex
noCollect:
    — need to sweep?
    bg,a done
    ld [%g_freeCons + %g_freeConsIndex], %result
    call lazySweep
    nop
    ld [%g_freeCons + %g_freeConsIndex], %result
done: ...

lazySweep:
    ...
    — %bitsLeft contains the remaining bits
    andcc %bitsLeft, 1, %thisBit
    bnz,a nextBit
    add %currentRef, ConsSize, %currentRef
    — sweep the word into the cons free-list
    st %currentRef, [%g_freeCons + %g_freeConsIndex]
    add %g_freeConsIndex, 4, %g_free_ConsIndex
    add %currentRef, ConsSize, %currentRef
nextBit:
    — on to the next bit
    srl %bitsLeft, 1,%bitsLeft
```

Figura 3.9 Lazy Sweeping por Zorn [JONES e LINS, 1999]

Através da técnica implementada por Zorn – chamada de *mark-and-lazy-sweep* – é possível perceber que o custo da varredura é contabilizado de acordo com o custo da alocação de novas células – relativamente pequeno; entre dez a doze ciclos.

3.8 Conclusões

Discute-se muito a respeito da comparação entre *mark-sweep* e os coletores por cópia. Muitos motivos – alguns descritos nos tópicos anteriores – levam a uma preferência pelo *mark-sweep*. Podemos citar, por exemplo, a questão do espaço e localidade, complexidade de tempo, dentre outros.

Um outro fator que pode ser determinante para a escolha do *mark-sweep* está associado a questão da mobilidade. Alguns sistemas não permitem a mobilidade de objetos na *heap* – ou seja, não permitem a alteração dos endereços dos objetos – o que inviabiliza os coletores por cópia [JONES e LINS, 1999].

Entretanto, o *mark-sweep* simples peca pelo fato de ser do tipo “pare o mundo”. Para contornar esse problema, utilizam-se, quando necessário, algumas das técnicas de implementação preguiçosas apresentadas neste capítulo.

Na prática, entretanto, observa-se que a fase de varredura (*sweep*) não é a mais custosa. Medições realizadas com a ajuda de *benchmarks* apontam um outro gargalo, o que motivou o desenvolvimento neste trabalho de modificações no algoritmo *mark-sweep* tornando-o *lazy*. Tais modificações serão abordadas no próximo capítulo.

4 Avaliação de Desempenho do Algoritmo Mark-sweep

Para avaliar os custos relacionados ao algoritmo mark-sweep no ambiente JVM, foram coletados os tempos de execução das suas quatro fases - marcação e reciclagem; computação dos novos endereços de memória dos objetos vivos; ajuste dos ponteiros e compactação - quando diferentes benchmarks são executados.

4.1 Experimentos com uma aplicação que utiliza lista encadeada

Uma estrutura de dados simples e que está presente em muitas aplicações é a lista encadeada. Nesta estrutura um objeto armazena, além de seus atributos, um ponteiro ou referência para um outro objeto do mesmo tipo ou sub-tipo.

No que diz respeito à coleta de lixo, é trivial observar que se nestas estruturas, em determinado momento da execução, existirem x objetos encadeados, e a aplicação mantiver referência ao objeto n, então existirão $x - n$ objetos presentes na memória que não serão coletados. Ao mesmo tempo, basta eliminar a referência ao primeiro (ou a qualquer outro objeto) para que todos os demais objetos encadeados a partir dele sejam considerados lixo.

Para avaliar os custos de coleta de lixo em listas encadeadas, neste trabalho foi desenvolvida uma aplicação Java relativamente simples. Os trechos de código relevantes são apresentados na figura 4.1.

O trecho (1) do código representa o objeto, de escopo estático, que mantém referência ao primeiro elemento da lista encadeada. Foi utilizado o modificador *static* para que, mesmo ao final da geração da lista, ainda exista uma referência da aplicação para ele.

O trecho em (2) apresenta o método chamado **gerarObjetos**, que tem por finalidade criar uma lista com *n* elementos. O valor *n* é lido na entrada e permite que listas encadeadas de diferentes tamanhos sejam geradas e avaliadas.

A classe que representa a lista encadeada é mostrada em (3). Ela contém apenas uma referência a um outro objeto do mesmo tipo e dois construtores.

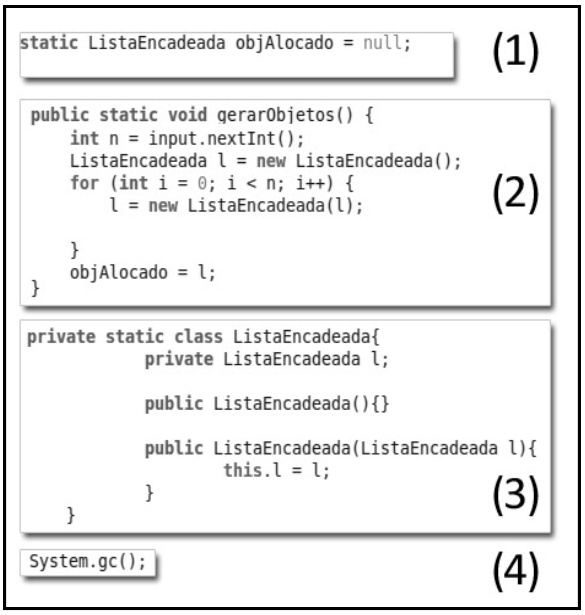

Figura 4.1 Principais trechos do *benchmark* desenvolvido

Java permite que a coleta de lixo seja disparada, de forma explícita, pelo programador. O mecanismo é ilustrado pela trecho de código (4). Assim, nem sempre é necessário esperar que a *heap* esteja cheia ou que um objeto muito grande precise de espaço para que a coleta de lixo seja disparada.

4.1.1 Resultados

Para que os tempos de cada fase que compõe o algoritmo de coleta de lixo pudessem ser coletados, foram inseridos alguns marcadores no código fonte da JVM (escrita na linguagem C++). Mais especificamente, foram introduzidas em cada uma das rotinas e sub-rotinas executadas pelo algoritmo de coleta de lixo instruções que medem o número de ciclos de *clock*

gastos em cada uma delas.

Em todos os testes apresentados nesta sub-seção, foram realizadas as seguintes ações: geração dos objetos, desalocação do primeiro objeto da lista e chamada explícita do coletor 10 vezes, calculando-se a média dos resultados. É interessante destacar que em alguns casos o resultado médio do número de ciclos de clock é zero devido à falta de precisão e arredondamento realizado pelo C++ no momento da impressão dos valores de cálculos envolvendo o número de ciclos de clock da máquina.

Inicialmente, foram gerados 1.000 objetos aninhados. Neste teste, verificou-se que a maior quantidade de ciclos foi gasta durante a execução das fases 1 e 3, com 10.000 ciclos cada uma, enquanto nas demais fases nenhum ciclo foi gasto.

Após um estudo mais minucioso das rotinas executadas durante a fase 1, que envolveu o refinamento da instrumentação da JVM, foi possível verificar a predominância do custo de execução das rotinas que percorrem os objetos com a finalidade de marcá-los.

À medida que se aumentou o número de objetos gerados de 10.000 para 100.000, observamos que cada uma das fases da coleta demandou 10.000 ciclos de clock.

Com 500.000 objetos pode-se observar que 30.000 ciclos foram gastos na fase 1, enquanto as fases 2 e 3 exigiram 20.000 ciclos cada. É interessante destacar que ocorreu uma coleta de lixo ainda durante a fase de alocação dos objetos, ou seja, antes que a coleta de lixo fosse explicitamente invocada, sem custos na fase 4.

Para 1.000.000 de objetos, de modo similar ao teste anterior, ocorreu uma coleta de lixo durante a fase de alocação da lista. Também manteve-se a quantidade de ciclos demandados pela fase 1. Entretanto, houve uma redução na quantidade de ciclos gastos nas fases 2 e 3, que passaram para 10.000, novamente sem custos na fase 4.

O coletor foi invocado 3 vezes ainda durante a geração da lista para o teste com 2.000.000 de objetos. Em cada coleta, a fase 1 demandou, respectivamente, 30.000, 20.000 e 40.000 ciclos. A quantidade de ciclos dispendidos nas fases 2 e 3 foi inferior, variando entre 10.000 e 20.000. A fase 4 não gastou ciclos. Terminadas as 3 coletas, a coleta explícita resultou em 10.000 ciclos de *clock* gastos para a primeira e terceira fases, e 40.000 para a segunda.

Por fim, quando 4.000.000 de objetos foram gerados, pode-se verificar nove chamadas ao coletor. Uma mensagem de erro de estouro de *heap* foi exibida na seqüencia, antes mesmo

que os objetos terminassem de ser alocados. Nas oito primeiras chamadas, a fase 1 sempre obteve ou um número de ciclos igual ou superior ao das demais fases. Já na última chamada, que antecedeu ao estouro de *heap*, a fase mais custosa foi a 2, com 30.000 ciclos de clock.

Para este primeiro *benchmark*, foi possível concluir que a fase 1, de marcação dos objetos, é a mais custosa, como indica a figura 4.1, onde os 'X's representam estouro de *heap*.

Lista Encadeada				
Num. Objetos	**Fase 1 (ciclos)**	**Fase 2 (ciclos)**	**Fase 3 (ciclos)**	**Fase 4 (ciclos)**
1000	10000	0	10000	0
10000	10000	10000	10000	10000
100000	10000	10000	10000	10000
500000	30000	20000	20000	0
1000000	30000	10000	10000	0
2000000 (1)	30000	10000	20000	0
2000000 (2)	20000	10000	20000	0
2000000 (3)	40000	10000	20000	0
2000000 (4)	10000	40000	10000	0
4000000	X	X	X	X
Média	**21111.11**	**13333.33**	**14444.44**	**2222.22**

Figura 4.1 Ciclos de clock gastos na coleta com Lista Encadeada

4.2 Experimentos com o *Benchmark* GCBench

GCBench é uma adaptação de Hans Boehm do *benchmark* de John Ellis e Pete Kovac. O *benchmark* cria e elimina árvores binárias de diferentes tamanhos, além de criar *arrays* de vida longa, ou seja, que são mantidos na *heap* e irão sobreviver às coletas enquanto não forem descartados.

De acordo com os comentários presentes no código-fonte do *benchmark*, não é possível simular situações reais de forma idêntica em *benchmarks*, entretanto, o GCBench é o que mais se aproxima destas situações.

Muitos são os fatores que implicam para a falta de exatidão para com os algoritmos encontrados em aplicações reais. São possíveis de serem citadas as deficiências presentes no GCBench – e que devem ser levadas em conta no momento de análise de seus resultados –

como: impossibilidade de se verificar o quanto de memória está em uso; não possuir estruturas de dados cíclicas; não ser possível medir variações de tempo ao passo que se variam os tamanhos dos objetos, e, por fim, os resultados também são custosos, ou seja, as rotinas de impressão dos mesmos consomem recursos do sistema segundo os comentários presentes no código-fonte do *benchmark*.

4.2.1 Resultados

Deve-se, antes de apresentar os resultados, destacar que, diferentemente do *benchmark* anterior, o GCBench não invoca o coletor de forma explícita. Isto é, todos os resultados apresentados são decorrentes de coletas de lixo disparadas pela própria JVM.

Inicialmente o *benchmark* tenta alocar uma árvore binária de profundidade 18. Durante esta alocação o coletor foi invocado. A primeira fase de coleta exigiu 20.000 ciclos de *clock*, enquanto as fases 2 e 3 consumiram 10.000 ciclos cada. Não se verificaram ciclos gastos na fase 4.

As rotinas seguintes do *benchmark* dispararam mais duas operações de coleta. Nestas coletas foram gastos 50.000 e 30.000 ciclos para a fase 1 e de 10.000 a 20.000 para as fases 2, 3 e 4.

Percebe-se, como no *benchmark* anterior que a primeira fase do algoritmo é mais custosa que as demais, como indica a figura 4.2.

	GCBench			
Rotinas	**Fase 1 (ciclos)**	**Fase 2 (ciclos)**	**Fase 3 (ciclos)**	**Fase 4 (ciclos)**
Árvore Binária	20000	10000	10000	0
Long-Lived Three Depth	50000	10000	20000	20000
Long-Lived Array	30000	20000	20000	10000
Média	**33333.33**	**13333.33**	**16666.67**	**10000.00**

Figura 4.2 Ciclos de clock gastos na coleta no GCBench

4.3 Java Grande Forum Benchmark Suite

Os *benchmarks* do pacote *Java Grande Forum Benchmark Suite – Version 2.0* são compostos por distintas aplicações e *kernels* que testam a JVM em uma série de situações presentes em programas reais [THE BENCHMARK SUITE, 2009].

Seus *benchmarks* são divididos em três seções: a) operações simples (operações aritméticas, chamadas de método e *casting*); b) *kernels* (pequenos trechos de código que são compostos por operações frequentemente utilizadas em aplicações); e c) aplicações de larga escala (códigos completos de aplicações, úteis) [THE BENCHMARK SUITE, 2009].

Para execução dos experimentos, foram utilizadas as mesmas instrumentações dos testes anteriores, sendo medidos os tempos de cada etapa do *mark-sweep*.

4.3.1 Resultados para as operações simples

Dentre as operações simples presentes na primeira seção do *benchmark*, é possível citar operações aritméticas, associações de diversos tipos, conversão de tipos, criação de objetos e *arrays*, chamadas a todos os métodos presentes em java.lang.Math (pacote de operações matemáticas disponível no Java), chamadas de método, serialização e tratamento de exceções.

Durante a execução de **JGFAll** – classe presente no *benchmark* que executa todos os *benchmarks* presentes nesta seção – todas as operações, exceto as de criação, duraram, em média, de 1 a 2 minutos e não forçaram o coletor da JVM uma única vez.

Entretanto, o *benchmark* **JGFCreateBench**, que testa o desempenho da JVM na criação de objetos e *arrays*, demorou cerca de 20 minutos para concluir sua execução e, ao longo desta, o coletor de lixo foi acionado incessantemente.

O que caracterizou todas as chamadas foi o predomínio do tempo gasto na primeira fase do algoritmo, a fase de marcação. Em todas as chamadas, o número de ciclos de *clock* resultante total da fase foi, em média, de 10.000.

4.3.2 Resultados para os *kernels*

Dentre os *benchmarks* desta seção, podemos citar os *kernels* que calculam os coeficientes de *Fourier* para a função $f(x) = (x+1)^x$ com intervalo 0,2; multiplicação de matrizes N x N; *heapsorts* (algoritmo de ordenação); criptografia; transformação unidimensional de N números complexos e cálculos com matrizes esparsas.

O *benchmark* possui como parâmetro o tamanho da entrada a ser utilizada quando da execução de cada um dos *kernels*. Os valores A, B e C, representam entradas pequenas, médias e grandes, respectivamente.

Após a execução do *benchmark* de tamanho A, as fases 2 e 3 foram as únicas que registraram custos, principalmente para os *kernels* de criptografia, ordenação e operações envolvendo matrizes. Para cada uma das fases citadas, o número de ciclos de *clock* gasto foi de no máximo 10.000.

Para a execução do *benchmark* de tamanho B as fases 2 e 3 novamente apresentaram os maiores custos para a maioria dos *benchmarks* avaliados. Porém para fatoração de sistemas lineares utilizando LU e para o *heapsort*, a fase 1 foi responsável pelo maior custo na coleta. Todos os valores coletados foram de 10.000 ciclos de *clock*.

Houve estouro de pilha logo no início da execução do *benchmark* de tamanho C, de forma que não foi possível avaliar os custos relativos a coleta de lixo neste *benchmark*. O estouro da *heap* ocorreu em virtude da grande exigência de memória do tamanho de entrada C. A figura 4.3 contém a listagem dos tempos calculados para as rotinas de *kernel* testadas, onde os 'X's representam estouro de *heap*.

Kernels				
Tamanho	**Fase 1 (ciclos)**	**Fase 2 (ciclos)**	**Fase 3 (ciclos)**	**Fase 4 (ciclos)**
A	0	10000	10000	0
B	10000	10000	10000	0
C	X	X	X	X
Média	**5000**	**10000**	**10000**	**0**

Figura 4.3 Ciclos de clock gastos na coleta de rotinas de *kernel*

4.3.3 Resultados para as aplicações de larga escala

O último pacote de *benchmarks* desta série é composto por algoritmos de busca (como exemplo, o jogo *connect4* em um quadro 6x7 usando a técnica de poda alfa-beta); solução de equações dependentes de tempo (Euler); modelos de partículas que interagem em um volume espacial sob condições periódicas fazendo uso do potencial de Lennard-Jones; simulações financeiras utilizando técnicas de Monte Carlo e por fim o cálculos dos caminhos de partículas em um sistema 3D.

Como na seção anterior, existem variações nos tamanhos das entradas, que desta vez são classificadas apenas em duas classes: A e B (pequenas e grandes entradas, respectivamente).

Durante a execução dos *benchmarks* de tamanho A, o coletor foi invocado com grande freqüência, sendo que a única fase a apresentar custos em todos os *benchmarks* foi a fase 1. Os custos desta fase variaram de 10.000 a 20.000 ciclos de *clock*. Para as demais fases do algoritmo de coleta de lixo não houve um padrão bem definido: ora apresentavam custos, que variaram de 10.000 a 40.000 ciclos, ora custo algum era registrado. Por fim, durante a execução do algoritmo de busca – que foi o último a executar – houve estouro da *heap*.

Apenas os algoritmos de Euler, Lennard-Jones e Monte Carlo executaram com o tamanho B. Todos os demais apresentaram estouro de *heap*. Nos *benchmarks* que executaram, verificou-se um custo maior durante a fase 3 (compactação), onde obteve-se, em praticamente todas as coletas, um custo entre 10.000 a 30.000 ciclos de *clock*. A fase 4 apenas em alguns poucos momentos apresentou custo, porém, de forma elevada, com valores entre 10.000 e 60.000 ciclos de *clock*.

Em suma, pode-se afirmar que os *benchmarks* presentes no pacote Java *Grande Forum Benchmark Suite – Version 2.0* tiveram resultados bastante semelhantes aos obtidos pelos demais *benchmarks*. Assim, praticamente em todos os casos, houve um custo maior na primeira fase do *mark-sweep*, salvo em algumas poucas exceções, como por exemplo na execução dos *benchmarks* de tamanho B em aplicações de larga escala.

A figura 4.4 lista os valores encontrados nas simulações, onde os ‘X’s representam estouro de *heap*.

Aplicações de Larga Escala					
Tamanho	**Algoritmo**	**Fase 1 (ciclos)**	**Fase 2 (ciclos)**	**Fase 3 (ciclos)**	**Fase 4 (ciclos)**
A	Busca	10000	10000	10000	0
	Connect4	20000	20000	10000	0
	Modelo de Euler	20000	10000	0	10000
	Lennard-Jones	10000	20000	20000	0
	Monte Carlo	20000	10000	10000	10000
	Partículas (3D)	20000	10000	40000	0
B	Busca	X	X	X	X
	Connect4	X	X	X	X
	Modelo de Euler	10000	10000	20000	60000
	Lennard-Jones	10000	10000	30000	0
	Monte Carlo	20000	0	20000	X
	Partículas (3D)	X	X	X	X
Média		**15555.56**	**11111.11**	**17777.78**	**10000.00**

Figura 4.4 Ciclos de clock gastos na coleta de Aplicações de Larga Escala

5 Simulações

Após realizadas as avaliações de desempenho do algoritmo *mark-sweep* e tiradas as conclusões a respeito dos custos de sua implementação na JVM, foi desenvolvida uma aplicação na linguagem Java para simular o funcionamento dos algoritmos *mark-sweep*, *mark-compact* (simples e com concorrência) e *mark-lazy-sweep*. O objetivo da implementação desta simulação é permitir uma comparação mais direta dos custos associados aos algoritmos. Com base nos resultados obtidos, eventualmente pode-se propor a implementação do melhor algoritmo de coleta na JVM.

Os passos seguidos para o desenvolvimento do simulador e os resultados obtidos serão descritos nos próximos tópicos.

5.1 Desenvolvimento da Heap

A estrutura principal do simulador que representa a *heap* é um vetor de *n* posições do tipo **Oop** (nome utilizado pela JVM para representar objetos). O vetor é estático e alocado uma única vez durante toda a execução do simulador.

Os métodos de alocação de objetos, assim como de liberação de memória, são públicos, isto é, são visíveis por outras classes, de forma a permitir que coletores, ou até mesmo outras estruturas, possam invocar tais ações. Ao mesmo tempo, métodos privados realizam toda a regra de alocação/desalocação de objetos, o que significa que qualquer estrutura tem o poder de solicitar alocação/desalocação, mas a forma como isso é feito é decidido pela *heap*.

Quando é solicitada a alocação de um objeto, a *heap* testa se há espaço disponível. Caso não haja, chama o coletor, que é apontado por um atributo presente na classe. O coletor também pode ser chamado, caso não haja espaço contíguo disponível, visto que todos os objetos são

alocados de forma contígua.

Para garantir que apenas uma *heap* seja utilizada por toda a aplicação, foi utilizado o padrão **Singleton** na classe. Este padrão é de simples implementação e caracteriza-se por declarar na classe a) um atributo estático de seu mesmo tipo, que será a única instância presente; b) um construtor privado (bloqueando a instanciação por classes externas) e c) um método estático que retorna a única instância da classe. Este último método, antes de retorná-la a instância, verifica se a mesma já foi criada, retornando-a neste caso. Caso contrário, realiza a instanciação antes de retorná-la.

5.2 Objetos

Oop foi o nome dado a um objeto no simulador, visto que a JVM faz uso deste mesmo nome para caracterizar o mais primitivo dos objetos (superclasse). Durante a simulação este é o único tipo de objeto existente no sistema.

Cada objeto possui um tamanho e uma lista de objetos aos quais ele aponta. Com isso, é possível simular as diversas formas como objetos interagem e mantêm ponteiros para outros objetos. Em programas reais, entretanto, ponteiros para os objetos são implementados de inúmeras formas, sendo a lista apenas uma forma de simplificar a visualização e execução da simulação.

No momento da coleta, o coletor faz uso da lista de objetos apontados para percorrê-los. O coletor faz uso ainda de um atributo do tipo `boolean`, utilizado como *bit* de marcação para que seja possível saber se o objeto está ou não em uso.

Na *heap* simulada os objetos são alocados em um vetor de tamanho pré-definido, devendo ser armazenados de forma contígua.

5.3 Coletores

Para que houvesse abstração quanto ao coletor e fosse possível armazenar, na *heap*, um

coletor independente de implementação, foi criada uma classe abstrata que possui diversos métodos de manipulação de objetos na *heap*. Seu único método público é o responsável por disparar a coleta.

Dessa forma, todos os coletores devem fornecer uma implementação desse método. Cada tipo de coletor é livre para implementar uma técnica diferente de coleta.

A partir deste momento, a *heap* então passa a contar com mais um atributo, o **coletor**, que pode ser alterado, inclusive, em tempo de execução, ou até mesmo mantido após o fim da coleta. Este é o caso do *mark-lazy-sweep*, que mantém um mapa de *bits*. O mapa mantido é gerado pelas operações de marcação. Uma vez gerado, pode ser utilizado pelas chamadas posteriores do algoritmo de coleta de lixo para evitar novas operações de marcação.

Os coletores também foram instrumentados com a finalidade de coletar os seus custos de execução. Entretanto, neste caso o tempo é medido em milissegundos.

Para o simulador, foram implementados os algoritmos *mark-lazy-sweep*, *mark-sweep*, *mark-compact* e o *mark-compact* concorrente, sendo os dois últimos mais próximos às características da implementação na JVM. A concorrência é um dos desafios na implementação de coletores, visto que uma série de cuidados devem ser tomados pelo simples fato de não ser possível prever todas as situações que podem ocorrer durante a coleta e a execução da aplicação em paralelo.

Ao final do capítulo, compararemos os custos dos algoritmos de coleta utilizando uma adaptação de alguns dos *benchmarks* apresentados no capítulo anterior.

5.4 Scripts

Durante a simulação e de forma semelhante a alguns dos *benchmarks* apresentados no capítulo anterior, os *scripts* executam a criação e destruição de objetos em larga escala, de forma a forçar chamadas aos coletores.

Todas as ações executadas pelos *scripts* são realizadas instanciando objetos **Oop** e utilizando a **Heap** criada para a simulação como local para armazenamento.

5.5 GUI

Para facilitar a visualização da *heap*, foi criado um componente gráfico para exibição dos seus espaços ocupados. A interface também fornece controles de criação de objetos, chamadas ao coletor e execução de *scripts*.

O conceito de herança foi aplicado à visualização da *heap*, visto que o componente herda JPanel e faz uso de seus métodos para exibir a ocupação dos objetos presentes nela através de retângulos e de outras formas geométricas fornecidas pela biblioteca.

A figura 5.1 apresenta a tela principal do simulador e destaca os seus principais controles.

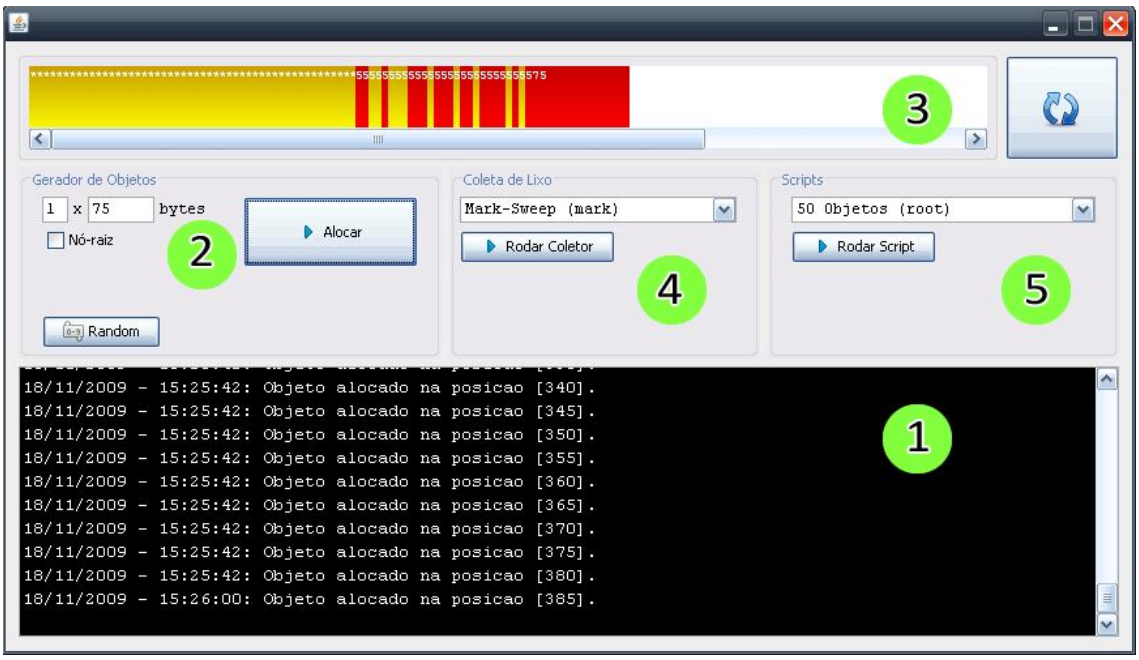

Figura 5.1 Tela principal do simulador

Em (1) encontra-se a saída de todos os controles presentes no simulador. Tanto a *heap* quanto os coletores podem emitir mensagens, que serão exibidas nesta área.

O gerador de objetos (2) permite a alocação de *n* objetos de tamanho *t* na *heap*.

A visualização da *heap* (3) permite um controle dos objetos em memória. Os blocos em

vermelhos representam, dependendo do momento: a) objetos alocados, ou b) objetos que não foram percorridos pelo coletor. Já os em amarelo são os objetos marcados pela fase de marcação do coletor (visto que todos os coletores implementados têm em comum esta fase). Os espaços em branco são locais ainda não ocupados – blocos livres.

Ainda no exemplo acima, a execução do coletor faria com que os objetos vermelhos fossem descartados, e os amarelos passassem a ficar vermelhos, visto que a fase de varredura desmarca todos os objetos marcados, como indica a figura 5.2.

Continuando na figura 5.1, em (4) são listados os coletores disponíveis, caso se deseje disparar o processo de coleta de forma manual. Para isso são listados todos os coletores implementados. A aplicação nesse caso tem um **Coletor** abstrato que só é instanciado nesse momento, mas disparado do mesmo modo, independente da implementação.

Por fim, em (5) é possível iniciar a execução de *scripts*.

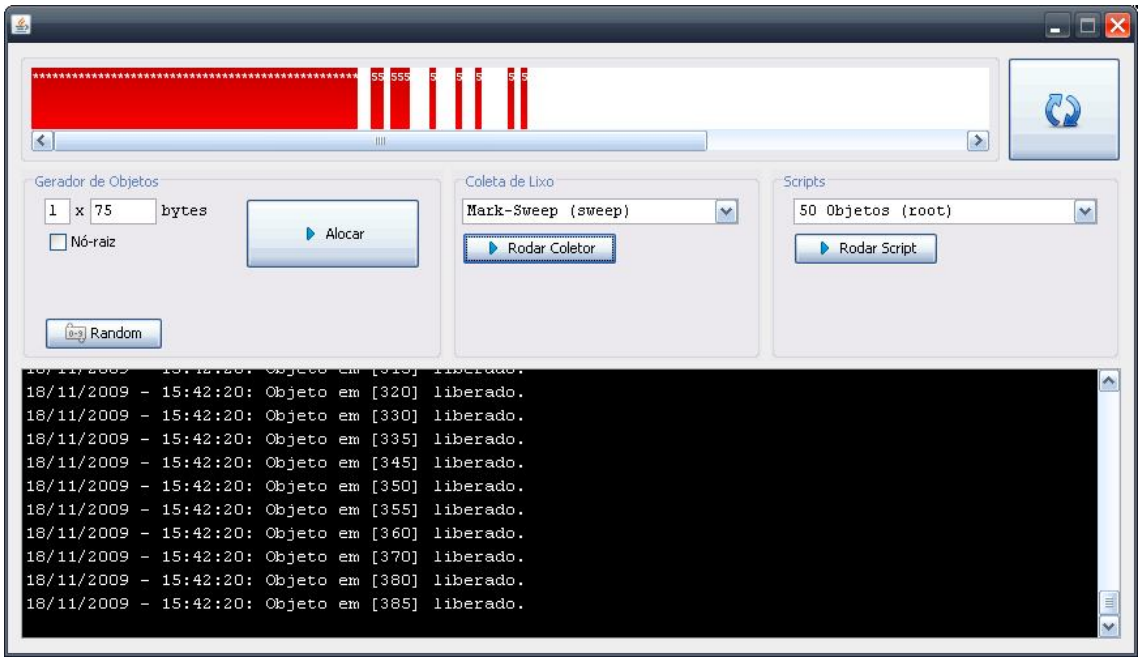

Figura 5.2 Simulador após a fase de varredura

5.6 Resultados das Simulações

Alguns dos *benchmarks* utilizados no capítulo anterior foram adaptados para serem executados no simulador para que pudessem ser analisados e comparados os custo relacionados as coletas dos três algoritmos estudados: *mark-compact*, *mark-compact concorrente* e *mark-lazy-sweep*. Este último é uma adaptação simplificada do *mark-sweep* para atender às características *preguiçosas* descritas no capítulo 3. Deve-se salientar que o mesmo não está presente na implementação da JVM.

O primeiro dos *benchmark* do capítulo anterior utilizados para avaliação de desempenho, lista encadeada, foi adaptado para o simulador mantendo-se suas principais características.

Quando executado na JVM, percebeu-se um custo maior na primeira fase do algoritmo, aspecto este que não se repetiu na simulação. Uma das possíveis explicações para este fato está relacionada a forma como os objetos são implementados na JVM real. A JVM é escrita fazendo uso de diversos padrões de projeto e dos conceitos de orientação a objetos. Isso facilita a manutenção e legibilidade do código, porém, é responsável, muitas vezes, por reduzir o desempenho. A implementação do simulador é muito mais simples, e portanto os custos para a marcação acabam sendo mais baratos do que na JVM real.

Este fato poderia levantar suspeitas quanto a corretude dos resultados de nossas simulações. Entretanto, como todos os algoritmos comparados fazem marcação, e todos utilizam o mesmo simulador com a simplificação dos objetos, acreditamos que os resultados podem ao menos apresentar uma tendência daquilo que ocorreria na JVM real.

Os maiores custos na simulação foram percebidos na fase 3 (compactação). Após 10 execuções com a *heap* totalmente ocupada, foram colhidas as diferenças de tempo e calculada a média dos valores, listadas na figura 5.3. Com isso, foi possível concluir que a diferença é de, aproximadamente, 10,53% a mais para o algoritmo em sua forma concorrente.

Execução	Mark-compact (ms)	Mark-compact Concorrente (ms)
1	2000	2400
2	2010	2040
3	2010	2100
4	2000	2450
5	2000	2010
6	2010	2400
7	2100	2100
8	2000	2300
9	2000	2450
10	2010	2010
Média	**2014**	**2226**
Diferença		**10.53%**
Desvio Padrão (mark-compact)	**29.05**	
Desvio Padrão (mark-compact concorrente)		**189.83**

Figura 5.3 Medições realizadas no simulador para lista encadeada

Os tempos medidos pelo *mark-lazy-sweep*, em comparação aos dois algoritmos listados na Figura 5.3, não foram apresentados na tabela por não fazer sentido compará-los em termos de tempo total, uma vez que seu funcionamento é completamente distinto. Não é possível simular o tempo de coleta da *heap* por se tratar apenas de uma varredura simples do bloco de memória, diferente da cópia, onde tomou-se 1 (um) milissegundo por *byte* copiado.

O algoritmo *mark-lazy-sweep* não copia objetos para outra *heap*, o que torna o tempo de varredura mais rápido que o *mark-compact*, não necessitando de nova marcação, caso a primeira marcação ainda mantenha, no mapa de *bits*, espaços contíguos para alocação de novos objetos.

Sendo assim, para aplicações com um grande número de alocação de objetos em um curto espaço de tempo, o uso coletor *mark-compact* pode resultar em muitas pausas, como visto na simulação, cujo tempo é proporcional ao tamanho total dos objetos ainda em uso. Essas pausas são amenizadas na presença do coletor concorrente, sendo que decisão de escolha entre o *mark-compact concorrente* e o *mark-lazy-sweep* deverá ser feita baseando-se na possibilidade de se trabalhar com paralelismo ou não entre o coletor e a aplicação. Já para determinados sistemas, pequenas pausas são aceitáveis, logo, o uso do *mark-lazy-sweep* é recomendável.

As árvores construídas pelo *benchmark* GCBench foram implementadas no simulador na forma de árvores binárias. Para isso, utilizou-se o recurso de armazenar referências de um objeto

para uma lista contendo dois objetos, simulando assim os nós esquerdo e direito de uma árvore binária.

Para não haver estouro de *heap* na execução da construção, a profundidade foi ajustada de modo a nunca estourar o tamanho total da mesma, sendo possível, desta forma, verificar o comportamento do coletor com a *heap* cheia.

Os tempos obtidos após a execução do coletor estão listados na figura 5.4.

Execução	Mark-compact (ms)	Mark-compact Concorrente (ms)
1	1050	1059
2	1042	1052
3	1029	1061
4	1028	1149
5	1098	1100
6	1030	1068
7	1029	1077
8	1029	1064
9	1030	1099
10	1029	1055
Média	**1039.4**	**1078.4**
Diferença		**3.75%**
Desvio Padrão (mark-compact)	**20.72**	
Desvio Padrão (mark-compact concorrente)		**35.77**

Figura 5.4 Medições realizadas no simulador para árvore binária

Para árvores binárias, a diferença de tempos de execução do coletor não é tão significativa como na simulação anterior; apenas 3,75% milissegundos a mais para o coletor paralelo, sendo que, em determinados casos, o tempo do coletor em série é praticamente o mesmo.

Novamente, o coletor m*ark-lazy-sweep* se enquadra como opção a aplicações que permitem um pequeno número de pausas, visto que a quantidade de pausas nele é menor que as do coletor concorrente por se restringirem apenas à fase de marcação. Neste caso, os tempos de varredura serão proporcionais ao tamanho de espaço que se deseja alocar.

Para árvores binárias, a fase de marcação do algoritmo *mark-lazy-sweep* demandou, em

média, 7,5 milissegundos após dez execuções do coletor disparadas manualmente, o que já mostra um custo maior nesta fase se comparada à implementação do *Mark-compact*. As fases de marcação dos dois algoritmos são implementadas de forma semelhante, exceto pelo fato do *mark-lazy-sweep* manipular um mapa de bits durante as iterações de marcação e que resulta no valor acima.

Aplicações que fazem uso de árvores binárias como principal estrutura de dados devem optar pelo *mark-compact* concorrente, por ter uma fase de marcação simples, sem necessidade de manipulação de mapas de bits.

A grande maioria dos algoritmos presentes no *benchmark* do *Java Grande Forum* são complexas, e não serão implementadas no simulador, com exceção da primeira seção de algoritmos, onde uma das rotinas era responsável por instanciar objetos de tamanhos pequeno, médio e grande. Tais instanciações eram gravadas na *heap* pela JVM (de forma implícita, ou seja, não é possível visualizar essa implementação no *benchmark*) que ocasionalmente realizavam chamadas ao coletor.

O **gerador de objetos**, ilustrado pelo item (2) da figura 5.1, é suficiente para realizar esta simulação, visto que o mesmo permite a criação de uma grande quantidade de objetos de tamanhos pré-definidos. Desta forma, pode-se simular uma situação semelhante às das aplicações reais, que muitas vezes fazem inúmeras instanciações de objetos de modo contínuo.

Para simular a instanciação de objetos pequenos, serão executadas dez simulações com 4096 objetos de tamanho equivalente a uma unidade de memória em uma *heap* capaz de armazenar 2048 unidades. Logo, algumas chamadas ao coletor serão realizadas ao longo do processo.

Execução	Mark-compact (ms)	Mark-lazy-sweep (ms)
1	12333	1593
2	20597	1610
3	8169	1593
4	18485	1710
5	30921	1656
6	30828	1563
7	31032	1547
8	12321	1531
9	20604	1594
10	12364	1562
Média	**19765.4**	**1595.9**
Diferença		**-91.93%**
Desvio Padrão (mark-compact)	**8226.27**	
Desvio Padrão (mark-compact concorrente)		**62.21**

Figura 5.5 Medições realizadas no simulador para instanciações simultâneas

O coletor *mark-compact* concorrente não foi incluído na comparação pois muitas chamadas conflitantes à cópia estavam sendo realizadas o que demandaria um controle complexo de *threads*, visto que a cópia é executada em paralelo. Caso fosse implementado, esse controle deveria ser semelhante ao controle exercido pela JVM. Como não foi possível estudá-la completamente de forma a se entender como esse controle é realizado, ele não está presente no simulador e permite apenas que objetos sejam alocados durante uma execução do coletor em paralelo.

As dez medições presentes na figura 5.5 representam os tempos de pausa gerados pelos coletores *mark-compact* e o *mark-lazy-sweep* durante as coletas que surgiram ao longo das instanciações.

O algoritmo *mark-lazy-sweep*, por sua vez, se mostrou eficiente, o que refletiu em um tempo extremamente curto para a instanciação dos objetos, cerca de 91,93% a menos que o *mark-compact*, como indica a figura 5.5. A fase de marcação, a princípio, também não gerou custos e manteve-se abaixo de 1ms.

Percebeu-se ainda nos benchmarks avaliados que esta variação do *mark-sweep* resultou em uma *heap* pouco fragmentada. Por trabalhar com alocação sob demanda, muitas vezes é possível “encaixar” o objeto antes de liberar novos espaços, ocorrendo a fragmentação somente

em casos onde a variação de tamanho dos objetos é grande. Esta característica que deve ser analisada durante a escolha do melhor algoritmo, visto que o *mark-compact* praticamente elimina a fragmentação da *heap*.

Após observar a execução do benchmark para objetos de tamanhos maiores, observou-se o mesmo comportamento. Como o simulador faz uma estimativa de tempo de cópia dos objetos, para copiar 1 bloco ele gasta 1 milissegundo. Sendo assim, objetos de tamanho 1024 serão copiados com aproximadamente 1024 milissegundos, da mesma forma que 1024 objetos de tamanho 1 também resultarão na mesma quantidade de tempo, e assim por diante para tamanhos maiores ou um número maior de objetos.

6 *Conclusões*

A idéia inicial do presente trabalho era a de propor melhorias nos algoritmos de coleta de lixo já existentes na JVM. A princípio, a idéia era substituir o algoritmo *mark-sweep* que se encontra implementado na mesma pelo algoritmo *mark-lazy-sweep*.

Após o código da JVM ser estudado, foi possível concluir que, apesar do conceito empregado no coletor padrão ser o do *mark-sweep*, na verdade se trata de uma adaptação com quatro fases do *mark-compact* – algoritmo que une o *mark-sweep* aos coletores por cópia. Dessa forma, implementar as características de uma fase *lazy* não seria possível já que a compactação não permite que a cópia "preguiçosa" seja feita sem um tratamento extremamente complexo (ou até mesmo impossível).

Somando-se a isso, após uma bateria de testes realizadas com os *benchmarks* descritos neste trabalho, percebeu-se que a necessidade maior não era a de realizar otimizações na fase de varredura, mas sim na de marcação, conforme os resultados apresentados no Capítulo 4. Essa tarefa se tornou bastante complexa pelo simples fato dos coletores de rastreamento trabalharem com objetos marcados e não-marcados para tirar conclusões a respeito do mesmo estar sendo usado ou não.

A melhoria da fase de marcação pode ser tema de trabalhos futuros e representaria um grande avanço a ser implementado nos algoritmos de coleta da JVM, assim como de outras máquinas virtuais e sistemas operacionais.

Com o estudo do código da JVM, foi possível encontrar dois tipos de coletores *mark-compact*. O primeiro deles, padrão, é do tipo pare o mundo, enquanto o segundo executa suas rotinas em paralelo com a execução da aplicação. As duas implementações são muito eficientes, deixando pouco espaço para a implementação de melhorias.

Desta forma, e tendo em vista o curto espaço de tempo disponível para fazer implementações, decidiu-se pelo desenvolvimento de um simulador capaz de encapsular rotinas de geração de objetos em uma *heap* virtual e que fosse adaptável a qualquer algoritmo. Desta

forma as modificações propostas nos algoritmos de coleta poderiam ser implementadas de forma simples e rápida.

O simulador foi utilizado para simular o funcionamento dos algoritmos de coleta presentes na JVM. Diferentemente do percebido quando da execução de *benchmarks* na JVM, observou-se que a fase de marcação, na verdade, é a menos custosa no *mark-compact*. Essa divergência de valores de tempo entre o simulador e a JVM pode ser explicada pela complexidade das estruturas de dados que representam os objetos na JVM real, bem como pelo emprego de padrões de projeto para percorrer os objetos alcançáveis. Como citado anteriormente, o emprego de padrões fornece uma legibilidade ótima do código, assim como facilita a manutenção do mesmo, porém, peca pelo desempenho. Desta forma, acreditamos que uma implementação mais simples dos objetos na JVM poderia melhorar significativamente o desempenho da coleta na JVM. A confirmação destas suspeitas poderia compor um trabalho futuro.

Por fim, com a execução do simulador, foi possível notar, em praticamente todas as estruturas testadas, que a velocidade com que o coletor concorrente realiza a coleta é menor do que a do coletor serial. Um dos fatores que pode justificar tal fato são as operações de sincronização realizados durante a cópia para que o que está sendo copiado não interfira no que está sendo instanciado paralelamente pela aplicação.

De qualquer forma, a cópia em paralelo e o *mark-lazy-sweep* são dois algoritmos que concorrem como melhor opção para todos os perfis de aplicação que utilizem as estruturas de dados presentes nos *benchmarks* utilizados por este trabalho. Para o algoritmo concorrente, o tempo de parada é extremamente reduzido se comparado ao *mark-compact*, restringindo-se somente à fase de marcação.

Já o *mark-lazy-sweep* é caracterizado por resultar em um tempo total reduzido se comparado aos coletores por cópia, visto que trabalham com a fase de marcação sob demanda e realizam uma varredura localizada, de acordo com a necessidade de espaço solicitada pela aplicação. Entretanto, a fase de marcação apresenta maiores custos à medida que o número de objetos presentes na *heap* são percorridos. Tais custos são resultantes da manipulação do mapa de *bits*.

Finalmente, como principal contribuição para futuros trabalhos, fica a idéia de melhoria, tanto do simulador, quanto no código da máquina virtual, do mark-compact seja

implementando as características lazy na primeira fase, seja procurando novas maneiras de efetuar a coleta de lixo como um todo.

Referências Bibliográficas

[BRUNO, Eric J. **G1: Java's Garbage First Garbage Collector.** Dr. Dobb's: 2009**.** Disponível em http://www.ddj.com/java/219401061?pgno=1. Acessado em agosto de 2009.]

[BOEHM, Hans-J. **Reducing Garbage Collector Cache Misses**. Hewlett-Packard: 2000.]

[BOEHM, Hans-J, WEISER, Mark. **Garbage collection in an uncooperative environment**. Software Practice and Experience. 1988.]

[FODERARO, John K., FATEMAN, Richard J. **Caracterization of VAX Macsyma.** Symposium on Symbolic and Algebraic Computation. ACM Press: 1981.]

[FODERARO, John K., SKLOWER, Keith, LAYER, Kevin, *et al*. **Franz Lisp Reference Manual**. Franz Inc.: 1985.]

[GARNER, Robin, BLACKBURN Stephen M., FRAMPTON, Daniel. **Effective Prefetch for Mark-Sweep Garbage Collection**. Departamento de Ciência da Computação. Australian National University: 2007.]

[GOETZ, Brian. **Java theory and practice: A brief history of garbage collection**. 2003. Disponível em http://www.ibm.com/developerworks/java/library/j-jtp10283/. Acessado em julho de 2009.]

[HUGHES, R. J. M. **A semi-incremental garbage collection algorithm.** Software Practice and Experience. Págs. 1081 – 1084. 1982.]

[JONES, Richard, LINS, Rafael. **Garbage Collection.** Algorithms for Automatic Dynamic Memory Management. Nova York: John Wiley & Sons, 1999.]

[LINDHOLM, Tim. YELLIN Frank. **The Java™ Virtual Machine Specification.** 1999. Disponível em http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html. Acessado em julho de 2009.]

[MCCARTHY, John. **Recursive functions of symbolic expressions and their computation by machine.** *Communications of the ACM.* 1960.]

[MORAES, Hérberte Fernandes. **Implementação de uma Heurística de Coleta de Lixo na Máquina Virtual Java**. Juiz de Fora: 2007. Monografia (Bacharelado em Ciência da Computação) – UFJF.]

[PRINTEZIS, Tony. **Garbage Collection in the Java HotSpot Virtual Machine**. Sun Microsystems: 2005. Disponível em http://www.devx.com/Java/Article/21977/1954. Acessado em agosto de 2009.]

[STEENKISTE, Peter. **Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization**. Tese de PhD (Universidade de Stanford). Califórnia. 1987.]

[SYBASE. **Chapter 3: Configuring Memory**. 2005. Disponível em http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.ase_15.0.sag2/html/sag2/sag274.htm. Acessado em julho de 2009.]

[THE BENCHMARK SUITE. **The Java Grande Forum Benchmark Suite**. 2009. Disponível em http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html. Acessado em novembro de 2009.]

[VENNERS, Bill. **Inside the Java Virtual Machine**. 1996. Disponível em http://www.artima.com/insidejvm/ed2/index.html. Acessado em julho de 2009.]

[ZORN, Benjamin G. **Comparative Performance Evaluation of Garbage Collection Algorithms**. Tese de PhD (Universidade da Califórnia). Março de 1989.]