Implementação de uma Heurística de Coleta de Lixo na Máquina Virtual Java

Héberte Fernandes de Moraes

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

Orientador: Prof. Dr. Marcelo Lobosco

Juiz de Fora, MG
Dezembro de 2007

Implementação de uma Heurística de Coleta de Lixo na Máquina Virtual Java

Héberte Fernandes de Moraes

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. Dr. Marcelo Lobosco – orientador
D.Sc. em Eng. Sist. e Computação,
UFRJ/2005

_____

Prof. Dr. Ciro de Barros Barbosa
D.Sc. em Computação**,**
University of Twente, UT,
Holanda/2001

_____

Prof. Dr. Rodrigo Weber dos Santos
D.Sc. em Matemática,
Physikalisch-Technische Bundesanstalt, P.T.B.,
Alemanha/2004

Juiz de Fora, MG
Dezembro de 2007

Agradecimentos

Gostaria de agradecer primeiramente à Deus, pelo dom da vida. Aos meus pais por sempre me apoiarem e acreditarem em mim. Aos meus irmãos e toda minha família. Aos meus colegas que sempre estiveram ao meu lado durante toda essa jornada, em especial ao Luciano pela ajuda durante este último período. Aos professores da UFJF pelo excelente trabalho realizado, em especial ao Marcelo Lobosco pelo incentivo e apoio, aos integrantes da banca examinadora por se prontificarem a participar da banca. Agradeço a todos que cruzaram o meu caminho durante todo esse tempo, talvez sem todos esses encontros meus esforços tivessem sido em vão.

Obrigado.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

O Coletor de Lixo (GC) foi criado com a finalidade de auxiliar o programador no desenvolvimento de suas aplicações. Em linguagens de programação que não contam com este recurso, o programador é obrigado a especificar explicitamente quais objetos devem ser liberados, o que em muitas situações pode provocar erros graves como, por exemplo, o término da aplicação por esta acessar incorretamente um endereço que não lhe foi alocado pelo sistema operacional. Com o recurso de GC, o programador fica livre dessa responsabilidade, levando a um gerenciamento mais eficiente da memória.

Ao realizar uma coleta de lixo, um custo é adicionado ao tempo total de execução do aplicativo. Para reduzir este custo adicional em termos de tempo de computação, podemos aumentar o tamanho do espaço disponível para o aplicativo realizar alocações. Por outro lado, disponibilizar memória em excesso para o aplicativo acarreta desperdício de recursos computacionais. Atingir um ponto de equilíbrio entre o custo da coleta de lixo e a quantidade de memória utilizada constitui-se, portanto, em uma tarefa árdua.

Neste trabalho apresentamos heurísticas que tem por objetivo ajustar, de forma automática, o tamanho da área disponível aos aplicativos para alocação de memória. Uma das heurísticas propostas foi implementada e avaliada na Máquina Virtual Java. Os resultados obtidos mostraram que a heurística implementada é efetiva para uma ampla gama de classes de aplicações avaliadas.

Palavras chave

Coleta de Lixo, Máquina Virtual Java, Heurísticas para Coleta de Lixo.

Capitulo 1

Introdução

O gerenciamento automático de memória vem sendo objeto de inúmeros estudos há algumas décadas. Várias linguagens de programação adotaram este mecanismo com o objetivo de retirar do programador a responsabilidade pelo gerenciamento do espaço de memória do aplicativo, reduzindo assim a possibilidade de ocorrerem erros durante este processo. Nestas linguagens, a figura do coletor de lixo personifica o papel de gerente da memória.

O trabalho do coletor de lixo se resume a vasculhar a memória, procurando por objetos que não podem mais ser acessados pela aplicação. Estes objetos são chamados de lixo de memória. A tarefa do coletor é retirar esses objetos da memória, liberando o espaço antes ocupado por estes para novas alocações. O coletor gerencia assim a alocação e a liberação de memória para o aplicativo.

O uso de coletores em linguagens de programação implica em um custo adicional para a aplicação, decorrente principalmente dos processos de coleta, pois parte do tempo de execução dos aplicativos é gasto nos processos de coleta de lixo. Esse custo pode ser muito alto, dependendo das características do aplicativo.

Neste trabalho apresentamos heurísticas para tentar reduzir o custo de coletas de lixo na JVM (*Java Virtual Machine*). Uma dessas heurísticas foi implementada, e seu desempenho foi comparado com os coletores presentes na JVM. Os resultados demonstram que a heurística foi efetiva em seu objetivo de reduzir os custos relativos à coleta de lixo.

O restante deste trabalho está assim estruturado. O capitulo 2 faz uma revisão das principais estratégias para coleta de lixo descritas na literatura, além de apresentar vários fatores importantes para a avaliação de um coletor de lixo. O capitulo 3 descreve a implementação dos coletores de lixo hoje disponíveis na JVM. O capitulo 4 apresenta as heurísticas propostas para reduzir o custo de coletas de lixo. O capitulo 5 avalia o desempenho da implementação de nossa heurística na JVM, comparando seu desempenho com outras implementações. Empregamos para os testes um conjunto representativo de aplicações, que apresentam diferentes padrões de utilização da memória pelas aplicações. O capítulo final apresenta nossas conclusões.

Capitulo 2

Coletor De Lixo

O coletor de lixo é um mecanismo com a função de eliminar memória não referenciada de forma automática. Com essa automação, o programador não precisa indicar cada eliminação, poupando esforços de programação e aumentando a segurança do código, uma vez que o processo manual é mais propenso a falhas. Algumas dessas falhas podem comprometer a estabilidade e levar ao término inesperado do aplicativo.

Essa maior confiabilidade e a grande eficiência – não só a referente ao gerenciamento da memória, mas também a eficiência produtiva emprestada à confecção do software – tornam o coletor de lixo uma necessidade em qualquer linguagem de programação.

A limpeza de regiões não referenciadas de memória deve ser um processo invisível, ou seja, os recursos do sistema não podem ficar reservados para a limpeza de memória durante muito tempo, pois o principal objetivo é alocar a maior quantidade possível de recursos para a execução do aplicativo. O dever do coletor de lixo é apenas de disponibilizar memória para o aplicativo, a não ser que toda ela esteja sendo utilizada, não havendo, portanto, lixo algum.

De acordo com (GOENZ, 2003), antes de conhecer os algoritmos de coleta de lixo, alguns critérios devem ser levados em conta:

- Tempo de espera: é todo o tempo que o aplicativo do usuário perde esperando pela da limpeza da memória. Esse tempo pode variar muito dependendo da estratégia escolhida para o coletor de lixo, mas o importante é que o tempo de execução do coletor seja o menor possível;
- Predição da pausa: as pausas para a execução do coletor de lixo devem ser feitas de forma que seja conveniente para o aplicativo e que não seja perceptível ao usuário;
- Tempo em “uso de UCP”: é o tempo que o coletor ocupa a UCP. É uma variável chave para o desempenho dos aplicativos, razão pela qual merece atenção especial. Esse tempo compõe em grande parte o tempo de espera;
- Divisão de memória: alguns algoritmos de coleta de lixo fazem a divisão da pilha de execução em espaços separados de memória para facilitar a coleta de lixo. Um exemplo desse algoritmo é a estratégia que divide a memória em dois espaços distintos:

enquanto um espaço está sendo utilizado normalmente pelo aplicativo, no outro espaço é realizada a coleta de lixo, sendo que tal processo necessita de exclusividade na utilização do espaço de memória em que trabalha. Dependendo da estratégia escolhida, uma das divisões pode se tornar inacessível ao programa para que o processo de coleta seja realizado, mas nunca todas as regiões podem ficar inacessíveis, pois senão o aplicativo não teria onde alocar ou acessar a memória;

- Interação com a memória virtual: deve-se ter uma preocupação maior em sistemas com pouca memória e que fazem uso constante de memória virtual, pois algoritmos completos de coleta de lixo (aqueles que varrem toda a memória) podem causar muitas falhas de página em sua execução, deixando o sistema ainda mais lento, já que o custo de uma falha de página é muito alto. Assim, seria desejável que o coletor gerenciasse corretamente as referências disponíveis na memória. Para isso o algoritmo deve manter informação relativa à localização de cada objeto, para que no processo de coleta ele possa minimizar as falhas de página verificando apenas as referências atualmente na memória principal. Com essa preocupação o algoritmo pode obter um melhor desempenho, pois tenta aumentar a taxa de *throughput* enquanto minimiza a paginação;
- Efeito sobre a localidade do programa: um coletor de lixo simples tem apenas o papel de recuperar memória não utilizada, mas isso nem sempre é o suficiente. Em algoritmos mais sofisticados, o coletor de lixo também é responsável por manter a localidade de referências do aplicativo, realocando objetos durante o processo de limpeza da memória de forma a manter próximos os objetos relacionados, o que pode colaborar para manter, ou mesmo melhorar, o desempenho do aplicativo;
- Ação do compilador e o impacto na execução: alguns algoritmos dependem da ação do compilador para poderem fazer seu trabalho. Cabe ao compilador adicionar código no aplicativo para auxiliar a tarefa do coletor. Esse código adicionado impacta a execução do aplicativo, uma vez que pode ser necessário executar instruções adicionais em algumas ocasiões. Devem-se conciliar as necessidades do coletor e de desempenho do aplicativo tendo em mente que quanto mais alterações sejam feitas pelo compilador no código original, possivelmente maior será o custo adicional na execução do aplicativo.

Um conceito importante e muito utilizado nos algoritmos de coleta de lixo é o de objeto. Objeto é um espaço de memória, com uma estrutura bem definida, e pode conter um ou mais campos, alguns dos quais podem conter referências. Uma referência é um endereço para um determinado objeto. Sempre que um objeto é criado uma referência a

esse objeto também é criada. O conceito de objeto, além de ser utilizado em linguagens orientadas a objetos, pode ser utilizado para descrever tipos estruturados, como por exemplo, os registros de dados (PRINTEZIS, 2004).

Na construção de um coletor, deve haver uma preocupação quanto aos tópicos discutidos anteriormente. Um algoritmo para a coleta de lixo deve, ao realizar a limpeza da memória, determinar quais áreas da memória estão reservadas para o aplicativo, mas que não são passíveis de acesso, ou seja, determinar quais são os objetos inalcançáveis e em seguida fazer a liberação dessa memória, disponibilizando assim esse espaço para uma futura utilização.

Objetos inalcançáveis são áreas de memória que o aplicativo não necessita mais, mas que estão reservadas para o aplicativo. Essas áreas da memória são chamadas de lixo de memória. A figura 1 ilustra uma pilha de execução com vários objetos alocados e lixos de memória.

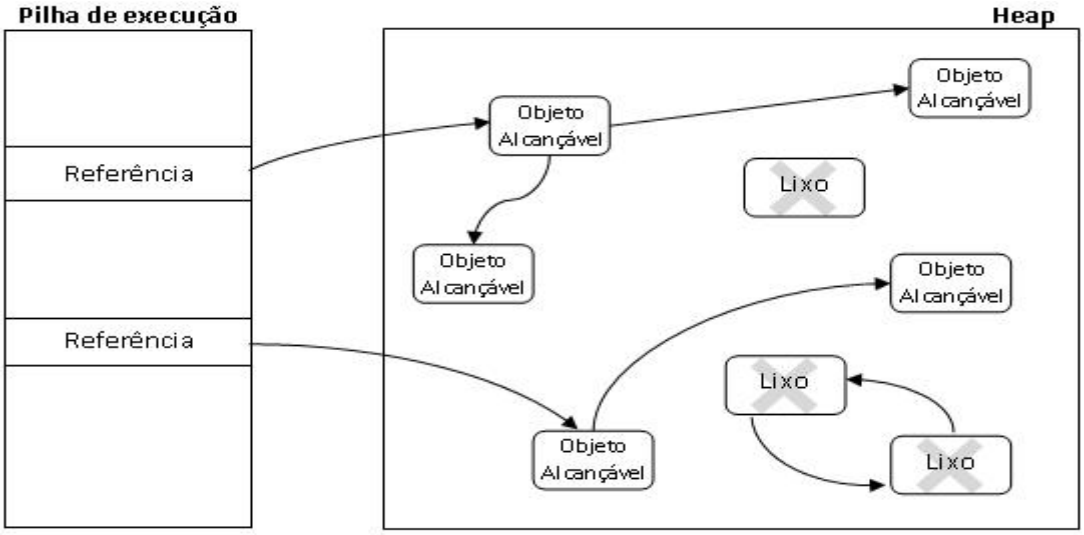

Figura 1 – Objetos alcançáveis e lixo de memória

O problema inicial de todos os algoritmos de coleta de lixo é o mesmo: como detectar os blocos de memórias alocados que são inalcançáveis pelo aplicativo do usuário. Várias estratégias foram criadas para realizar essa tarefa, algumas delas estão descritas neste trabalho.

2.1. Contagem de Referencia (Reference Counting)

De acordo com (VENNERS), esse método de coleta de lixo é o mais simples, mas necessita de auxilio do compilador. Cada objeto do aplicativo mantém um contador de

referências. O contador deve ser incrementado a cada nova referência a esse objeto e decrementado caso uma dessas referências deixe de existir ou se a referência a esse objeto for reciclada. Quando o número de referência é igual a zero, então esse objeto é reciclado. O compilador é responsável por adicionar o código para manipulação dos contadores de referências.

De acordo com (Levine & Young, 2007), esse algoritmo é uma estratégia de gerenciamento de memória automático que pode ser implementada utilizando estruturas de dados simples, com operações simples e de baixo custo na execução. Apesar da simplicidade operacional e de baixo custo é difícil criar uma implementação eficiente por causa da freqüência em que essas operações ocorrem. Em ambientes em que muitos objetos e várias referências são criados, a aplicação desse algoritmo de coleta de lixo é impraticável, pois o custo total de atualização dos contadores de referências seria muito alto.

Também é difícil ter confiabilidade com esse algoritmo, pois essa estratégia não consegue fazer a coleta em objetos com referências cíclicas. Outras estratégias que se baseiam nesta técnica foram criadas tentando atacar os seus pontos falhos.

2.1.1. Contagem de Referência (Deferred Reference Counting)

Esse algoritmo é uma variação do *reference counting*. Durante o processo de coleta de lixo esse algoritmo analisa um número reduzido de referências, o que pode levar ao aumento do desempenho do mesmo. Enquanto outros algoritmos analisam todas as referências do programa, este analisa somente a pilha de execução, onde estão localizadas as variáveis locais aquele ponto de execução.

Como o espaço de contagem de referências é reduzido, um objeto não pode ser reciclado quando seu número de referências chega à zero, pois pode haver alguma variável do programa que faça referência a esse objeto. Em contrapartida as variáveis do programa são periodicamente verificadas. Um objeto é reciclado somente quando no processo de contagem não existe nenhuma referência a ele, ou seja, quando na pilha de execução nenhuma variável mantém ligação com este objeto e no processo de verificação das variáveis do aplicativo também não há nenhuma referência a este objeto.

Esse algoritmo tem bons resultados quanto a liberação de memória, pois de acordo com (RAVENBROOK 2001), a maior parte das referências a objetos são

provenientes das variáveis locais. Além do mais, o custo para a contagem de referências é reduzido.

2.1.2. Contagem de Referências (One-bit Reference Counting)

Essa é outra variação do *reference counting* que consiste em manter um único bit em cada objeto para indicar se este possui uma (valor zero) ou várias referências (valor um) a ele. Apenas quando o bit vale zero, indicando que o objeto é referenciado apenas uma vez, podemos incluí-lo no processo de reciclagem. Logo objetos cujos bits valem um nunca serão reciclados.

O processo de contagem de referências é semelhante ao utilizado pelo *reference counting*. Um objeto tem o seu bit iniciado com o valor zero. Caso existam duas referências a este objeto, o bit é alterado para o valor um e o processo de contagem para este objeto se encerra. Quando o bit passa para o valor um, o objeto em questão é descartado do processo de coleta de lixo, por ser considerado um objeto alcançável, ou seja, o bit nunca volta para o valor zero.

Apenas os objetos com o bit indicando o valor zero participam do processo de coleta de lixo. A contagem de referências é reduzida, pois não há a necessidade de saber a quantidade total delas para um determinado objeto, mas apenas se há referências simples ou múltiplas. Como o principal objetivo dessa estratégia é o de reduzir o conjunto de objetos a serem analisados, nesse conjunto reduzido de objetos podem ser aplicadas outras técnicas para realizar a coleta de lixo propriamente dita.

Esse algoritmo é mais rápido que o *reference counting*, seu bom desempenho é devido, de acordo com (RAVENBROOK 2001), a maioria dos objetos tendem a ter apenas uma referência para si durante todo seu ciclo de vida, com isso evita-se o desperdício de processamento na contagem das referências, pois é necessário saber somente se existe pelo menos uma referência a este objeto.

2.1.3. Contagem de Referência (Weighted Reference Counting)

Essa técnica é usada para coleta de lixo em ambientes distribuídos por causa da baixa taxa de comunicação inter-processos.

No algoritmo *weighted reference counting* cada objeto tem um contador de referências, como no algoritmo de *reference counting*. Para cada referência, além do endereço do objeto, esse algoritmo contém um campo para armazenar um peso. Esse

campo irá armazenar uma fração do valor total contido no campo contador de referências no objeto de destino.

Quando um objeto é criado assume-se que existam várias referências para este objeto. Assim, um valor arbitrário é associado ao contador de referências (restringido a uma potência de dois), e à primeira referência a esse objeto é dado um peso igual ao do contador de referências do objeto.

A soma dos pesos de todas as referências para um objeto sempre será igual ao valor associado ao seu contador de referências. Para se criar novas referências a um objeto, um processo de cópia é realizado: a partir de uma referência já existente, uma cópia é gerada e o peso da referência pai é dividido igualmente entre ambas. Nesse processo não há a necessidade de alterar o campo contador de referências do objeto, atendendo ao objetivo inicial da diminuição da troca de mensagens.

Quando uma referência é apagada, uma mensagem contendo o peso da referência em questão é enviada para o objeto e esse peso é decrementando no valor total contido no campo contador de referências. Essa é a única comunicação necessária, por isso a quantidade de mensagens trocadas é reduzida. O coletor é responsável por receber as mensagens para os objetos e atualizar o campo de contagem de referências. Sempre que o valor se torna zero, o objeto em questão é reciclado. O processo de comunicação é assíncrono, pois não há necessidade de sincronização entre os processos, além do que esse algoritmo funciona com qualquer protocolo de comunicação por não necessitar de ordenação das mensagens.

2.2. Algoritmos de Rastreamento (Tracing Collector)

Os algoritmos *tracing collector* consiste em percorrer todas as referências a partir do conjunto inicial de referências denominado raiz. Tal conjunto contém os espaços de memória que não serão reciclados pelo coletor. Esse conjunto raiz pode ser composto pelas variáveis do programa do usuário, parâmetros da função em execução, variáveis globais, dentre outros. Todas as referências alcançáveis direta ou indiretamente a partir desse conjunto permanecerão intocadas, enquanto que as referências inalcançáveis serão recicladas. O conceito de alcançabilidade é transitivo, ou seja, se uma referência A é dita alcançável e a partir de A chega-se em B, B também é alcançável.

Todas as variantes deste algoritmo baseiam-se no *tri-colour marking*, que consiste em criar três conjuntos nomeados como conjunto branco, cinza e preto. O conjunto branco

consiste de todos os objetos que são candidatos a serem reciclados. No conjunto preto estão as referências que não se referem a nenhum objeto do conjunto branco. O conjunto cinza é o intermediário entre o grupo branco e preto, onde estão os objetos alcançáveis já marcados, mas cujos descendentes ainda não foram verificados.

Esse algoritmo funciona da seguinte forma: um objeto do conjunto cinza é movido para o conjunto preto; todos os objetos do conjunto branco referenciados diretamente por este são movidos para o conjunto cinza. Esse processo é repetido até que o conjunto cinza fique vazio. O algoritmo termina quando não há mais objetos no conjunto cinza, então todos os objetos que restarem no conjunto branco são considerados inalcançáveis, podendo assim, ser reciclados.

Esse algoritmo é o mais utilizado e, ao contrário do *reference counting*, é capaz de fazer a coleta em objetos com referências cíclicas, o que o torna mais confiável. Esse tipo de estratégia verifica exatamente quais referências potencialmente serão utilizadas pelo aplicativo a partir do conjunto raiz de objetos, diferentemente dos algoritmos discutidos anteriormente.

2.2.1. Coleta por marcação e varredura (Mark-sweep Collection)

É o mais simples dos *tracing collectors*. De acordo com (GOENZ, 2003), este foi o primeiro algoritmo de coleta de lixo, projetado por *McCarthy* para a linguagem Lisp. Esse algoritmo se divide em duas fases (figura 2). A primeira fase consiste da marcação dos objetos alcançáveis a partir do conjunto raiz, e a segunda fase consiste da verificação que percorre todas as referências e recicla as não marcadas.

Esse algoritmo tem alguns inconvenientes:

a) A memória deve ser verificada por completo, fase em que o algoritmo realiza a marcação dos objetos alcançáveis, antes de realizar a liberação de qualquer objeto no espaço. Pois o tempo de pausa do aplicativo é proporcional ao tempo em que o coletor perde verificando toda a memória. Quanto maior a memória utilizada maior será o tempo de pausa;

b) A execução do algoritmo deve ser completa, caso haja alguma interrupção, deve-se iniciar o algoritmo desde o inicio. Portanto, esse algoritmo, na sua forma original, não é recomendável para aplicações de tempo real, necessitando assim de outras estratégias mais sofisticadas de coleta de lixo, que podem ser derivadas desta;

c) Outro problema é a fragmentação de memória, que pode levar a falhas de alocação mesmo que haja memória suficiente disponível, mas não de forma contígua.

A fragmentação de memória pode provocar a perda de desempenho do aplicativo por quebrar o princípio da localidade de referências: objetos que deveriam estar próximos na memória por pertencerem a um mesmo fluxo de execução podem estar em diferentes áreas ou até mesmo alocados em disco. Quando esse fluxo de execução entrar em operação, será necessário que todos os seus dados estejam novamente disponíveis na memória. Isso pode provocar, em ambientes com pouca memória e que utilizem memória virtual, trocas excessivas de páginas decorrentes dos seguidos acessos a páginas alocadas em disco que contenham os objetos requisitados.

Figura *2* – Algoritmo *Mark-sweep*

2.2.2. Coleta por cópia (Copying Collection)

Após o processo de coleta de lixo, muitas vezes ocorrem erros de alocação mesmo havendo memória disponível, pois vários algoritmos de coleta de lixo deixam a memória fragmentada após este processo. Visto que a fragmentação pode reduzir o desempenho do aplicativo pelo fato de não manter a localidade das referências.

O algoritmo *copying collection* se propõe a resolver esse problema realocando os objetos alcançáveis, corrigindo todas as possíveis referências para esses objetos e reciclando os demais, como ilustrado na figura 3. Pelo fato de evitar o problema de fragmentação, essa estratégia pode ser combinada com outras técnicas de coleta de lixo, como o *Mark-sweep*. O algoritmo consiste em dividir a memória em dois semi-espaços,

um que contém os dados ativos e outro com os dados não usados. Quando o espaço de dados ativos fica cheio, o aplicativo pára. Os dados ativos são copiados para o espaço de dados inativos. E então os papéis são trocados, ou seja, o espaço de dados ativos se torna o espaço de dados inativos e vice-versa.

Esse algoritmo também tem algumas desvantagens. Apesar de ser uma boa técnica, o custo pode ser elevado em virtude das operações de cópia e realocação de dados, além de necessitar de espaço extra para a realocação dos objetos.

Figura 3 – Algoritmo *copying collection*

2.2.3. Coleta por marcação e compactação (Mark-compact Collection)

Esse algoritmo é a união de dois outros algoritmos, o *mark-sweep* com o *copying colletor*. Ele une a estratégia de marcação de verificação do *mark-sweep* com a vantagem de não deixar a memória fragmentada após a execução do coletor do *copying colletor*. Como ilustrado na figura 4, essa estratégia consiste em realizar o marcar os objetos vivos e compactar a memória após o processo, realocando os objetos e alterando as referências.

Esse algoritmo aumenta o desempenho do aplicativo pelo fato de manter a localidade das referências, apesar de seu custo adicional decorrente da realocação dos objetos e troca das referências. Outra vantagem é que o algoritmo evita a ocorrência de erros de alocação decorrentes da fragmentação da memória.

Figura 4 – Algoritmo *mark-compact*

2.2.4. Coletor incremental (Incremental Collection)

Algoritmos de coleta de lixo do tipo incremental têm a capacidade de pausar no meio de sua execução, ao contrário de alguns algoritmos que exigem execução por completo sem interrupções. Os algoritmos incrementais permitem que, após uma pausa, continue-se posteriormente a execução sem que ocorram inconsistências de dados no final do processo.

A idéia é executar uma série de pequenos passos enquanto o programa executa, sem que sejam necessárias paradas muito longas. Ao construir um algoritmo deste tipo deve-se assegurar que o aplicativo não interfira na coleta e vice-versa; por exemplo, quando o programa necessita alocar um novo objeto, o sistema pode suspender o pedido até que o ciclo de coleta esteja completo, ou de algum modo notificar o coletor que existe um novo objeto alcançável.

O aplicativo e a coleta de lixo executam de forma alternada sem pausas muito demoradas e sem desperdício de esforço. Alguns algoritmos derivados do *tracing collector* são incrementais como, por exemplo, o *mark-sweep*, que é dividido em duas fases, sendo que cada fase pode ser intercalada com a execução do aplicativo.

Uma situação que deve ser tratada é a alteração, em tempo de execução, do grafo que representa os objetos na memória. Para resolver este problema em um esquema incremental deve haver algum meio de manter o coletor a par das mudanças no grafo de objetos alcançáveis, talvez re-computando as regiões em que as mudanças aconteceram.

2.2.5. Coletor conservador (Conservative Garbage Collection)

O algoritmo de coleta de lixo básico, como discutido no início deste capítulo, consiste em varrer toda a memória em busca de objetos inalcançáveis. Nesta forma o algoritmo é difícil implementar, já que a maioria das linguagens não prove ao coletor informação sobre os tipos de dados. Com isso o coletor não consegue distinguir o que é referência do que não é.

Uma técnica conhecida como *conservative garbage collection* foi criada com este problema em mente: a partir de um conjunto raiz de objetos, ela considera todos os campos de cada objeto como uma referência. Objetos referenciados pelo conjunto raiz não são reciclados, caso as referências sejam válidas. A transitividade é válida, ou seja, os objetos que são referenciados pelos objetos já referenciados também não são reciclados.

Pelo fato do algoritmo não saber distinguir com certeza onde, em um objeto, estão localizadas referências para outros objetos, ele não pode ser combinado com o *copying collection*, pelo fato de que este algoritmo necessita conhecer exatamente quais são as referências para realizar a realocação e corrigir estas referências.

Esse algoritmo pode, a primeira vista, parecer não ter muita eficácia, deixando de coletar muitos objetos, mas na prática ocorre o contrário. De acordo com (RAVENBROOK, 2001) esse algoritmo é bem efetivo e existem refinamentos que o tornam ainda melhor.

2.2.6. Coletor por geração (Generational Garbage Collection)

Essa estratégia surgiu a partir da observação de que a maior parte dos objetos tem um tempo de vida muito curto. Essa informação é útil para restringir a ação do coletor de lixo em objetos recentemente alocados.

A área azul da figura 5 é uma distribuição típica do tempo de vida dos objetos. O eixo X representa o tempo de vida dos objetos em bytes alocados. O eixo Y representa o total de bytes nos objetos. O pico à esquerda representa a quantidade de objetos foram alocados, mas que com o tempo foram coletados. Isso ocorre, por exemplo, em longas interações com objetos como em laços (*loops*), onde alguns objetos morrem após a execução do laço.

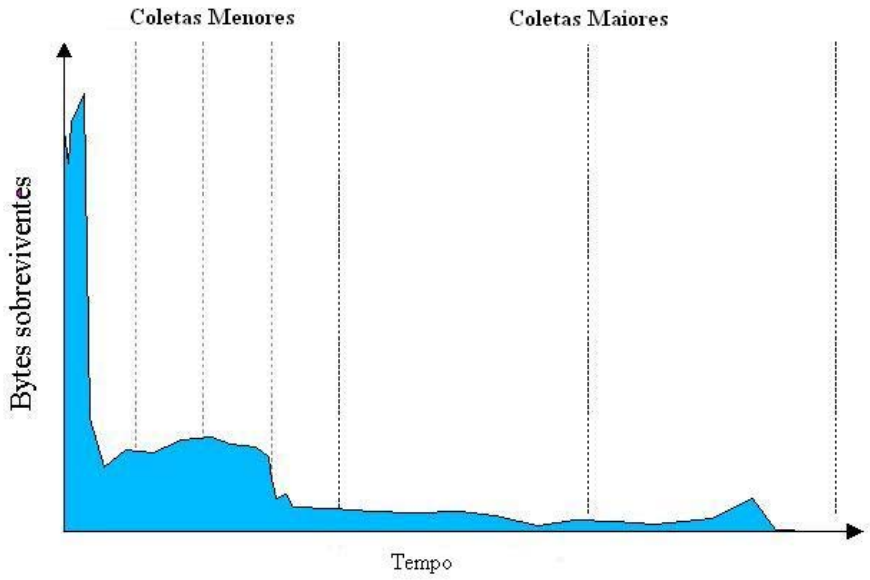

Figura 5 – Bytes sobreviventes ao decorrer da execução das coletas de lixo

O *generational garbage colletion* consiste em separar os objetos em gerações. Cada geração tem seu próprio espaço na memória. Objetos novos são colocados nas gerações mais jovens. Quando o espaço de alocação daquela geração se acaba, o coletor executa a limpeza de memória verificando as referências a partir de um conjunto raiz de objetos. Objetos considerados inalcançáveis são reciclados. Após várias interações desse algoritmo os objetos que continuam alocados são promovidos, ou seja, copiados para a próxima geração. Quando as gerações mais antigas ficam cheias, essas e todas as gerações mais novas são coletas. A figura 6 ilustra esse processo.

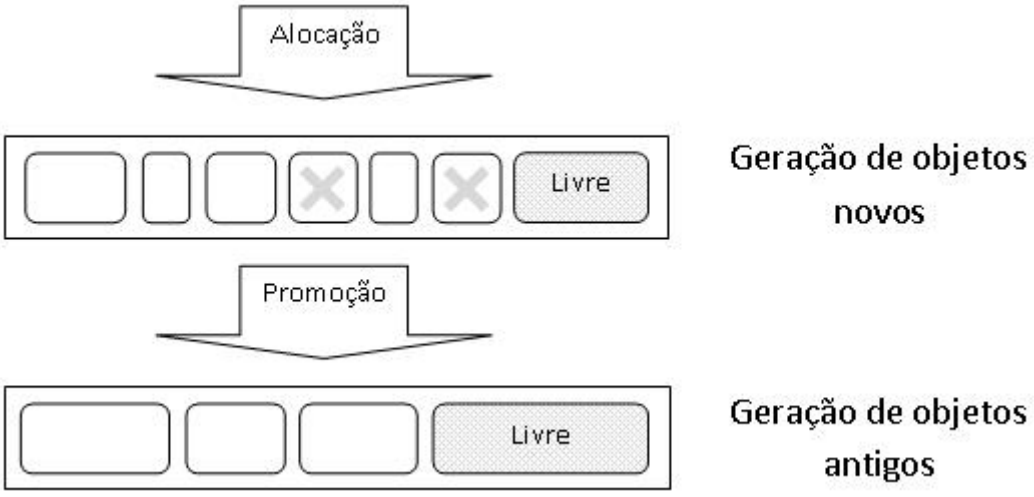

Figura 6 – Algoritmo *generational collection*

Com essa estratégia alguns objetos inalcançáveis podem não ser reciclados em cada ciclo, porque como o algoritmo é menos executado em gerações mais antigas, alguns objetos que pertencem a essas gerações podem se tornar inalcançáveis e continuar alocados. Para evitar essa situação, de tempos em tempos outros algoritmos, como o *mark-sweep*, podem ser executados nessas gerações para fazer a reciclagem destes objetos.

Na verdade muitas linguagens utilizam versões híbridas deste algoritmo. Em algumas versões dessa estratégia é possível executar diferentes algoritmos de coleta de lixo em gerações distintas. Essa propriedade é uma grande vantagem em relação aos outros algoritmos, pois assim pode-se dar um tratamento diferenciado a cada geração. Em gerações mais novas podem-se executar algoritmos mais simples e rápidos pelo fato desta ser a geração que terá a maior quantidade de objetos reciclados. Caso nenhuma memória venha a ser recuperada no processo, algoritmos mais sofisticados podem ser executados nessa mesma geração ou em gerações mais antigas. No fim, caso nenhuma das gerações tenha memória disponível para reciclagem, um algoritmo completo de coleta de lixo pode ser executado.

2.2.7. Coletor paralelo e concorrente

Os algoritmos descritos anteriormente foram elaborados para ambientes monoprocessados e a maioria destes necessita de todo o poder computacional alocado para o algoritmo, mesmo que não seja totalmente utilizado. Em ambientes multiprocessados estes algoritmos podem provocar desperdício. Para resolver este problema de desperdício de poder computacional, algumas novas estratégias foram elaboradas, tentando adaptar alguns deste algoritmo descritos anteriormente em ambiente com vários processadores.

Uma das estratégias é conhecida como *parallel collector*, que consiste em dividir as tarefas do coletor em partes independentes que possam ser executadas simultaneamente. Essa técnica pode ser associada ao *mark-sweep*, dividindo cada fase em processos separados para serem executados em processadores diferentes. Pode ser associado também ao *copying collector*, dividindo a pilha de execução em vários pedaços, sendo que o processo de verificação de cada pedaço é associado a um processador. Sua aplicação é ideal em ambiente com multiprocessadores, pois cada tarefa será alocada em um processador diferente.

Outra variação é conhecida como *concurrent mark-sweep collector*, versão que executa de forma parcialmente concorrente. Funciona da mesma forma que o *mark-sweep*,

com uma pequena diferença: o algoritmo pára momentaneamente a execução do aplicativo na fase de marcação e remarcação, já na fase de verificação o coletor executa de forma concorrente ao aplicativo. Esta variação é utilizada em gerações mais antigas no *generational garbage collection.*

Esses dois algoritmos nada mais são do que versões aprimoradas de outros algoritmos simples, para ambientes com multiprocessadores, de forma a obter menos desperdício de recursos e maior desempenho.

Capitulo 3

Mecanismos de Coleta de Lixo da Máquina Virtual Java

Java é uma linguagem orientada a objetos, característica baseada nas linguagens Smalltalk e Simula67, com a sintaxe similar a linguagem C/C++, distribuída com um vasto conjunto de bibliotecas (ou API's). Essa linguagem foi criada para ser independente de plataforma, para que isso aconteça foi desenvolvida uma máquina virtual. A portabilidade dessa linguagem é alcançada em arquiteturas que tenham a máquina virtual disponível para instalação.

A máquina virtual java (JVM – *Java Virtual Machine*) é um conjunto de aplicativos e estruturas de dados que implementam um modelo de máquina virtual especificado pela Sun, criadora da JVM. Esse modelo aceita uma linguagem intermediária que é conhecida como java *bytecodes*. O conjunto de instruções desses *bytecodes* é dividido em grupos de tarefas, como: instruções de carregamento e armazenamento, instruções aritméticas, conversões de tipos, criação e manipulação de objetos, instruções de manipulação da pilha de execução (push/pop), transferência de controle, invocação e retorno de métodos, lançamento de exceções e sincronização de processos (ou *threads*). O objetivo dos *bytecodes* é a compatibilidade binária.

A maioria das tentativas de melhorar o desempenho da linguagem java está em aplicar novas técnicas de compilação na JVM. Uma técnica muito interessante e eficaz é conhecida como compilação j*ust-in-time* (JIT), que consiste em traduzir cada parte dos *bytecodes* em código nativo de máquina na primeira vez em que for chamado. A JIT é quem realmente executa os *bytecodes* e compila cada parte do aplicativo na primeira vez em que é executado. Essa foi a idéia que trouxe melhores resultados na questão de desempenho.

Os aplicativos java são traduzidos para *bytecodes*, para executá-los os usuários necessitam ter a JVM instalada em seu sistema, pois os *bytecodes* são compilados pelo JIT contido na JVM e convertidos para código executável de máquina compatível com a plataforma em que a JVM está em execução.

No processo de execução dos *bytecodes*, a JVM realiza otimizações de acordo com o tipo especificado. A JVM tem implementado dois compiladores de *bytecodes*, o *client compiler* e o *server compiler* (Microsystems, 2006).

O *client compiler* é simples e dividido em três fases, a primeira fase que é independente de plataforma, consiste em criar uma representação intermediária de alto nível dos *bytecodes*, a partir dessa representação intermediária é possível realizar algumas otimizações no código. Na segunda fase, dependente da plataforma, é gerado outra representação intermediária de baixo nível. A última fase consiste em fazer a alocação de registradores e a geração de código de máquina, algumas otimizações também são realizadas nessa etapa (Microsystems, 2006).

O *server compiler* é mais complexo e está ajustado para obter o maior desempenho, esse é o perfil típico de aplicativos que rodam em servidores. Este compilador foi designado para fazer o maior número de otimizações possíveis. Ele executa todas as otimizações clássicas e também otimizações mais especificas a tecnologia java, além de ser altamente portável (Microsystems, 2006).

A principal diferença entre ambos os compiladores presentes na JVM é que no *client compiler* tem como objetivo minimizar o tempo de início do aplicativo e a memória utilizada, enquanto que o *server compiler* tem como objetivo maximizar a execução do aplicativo. O compilador padrão da JVM é o *client compiler*, mas é possível fazer o uso do *server compiler* adicionando a opção *-server* na JVM quando for realizar a execução de algum aplicativo.

Outra característica importante é a desalocação automática de memória, através do processo de coleta de lixo. Para ajudar no processo de coleta de lixo, todos os objetos são alocados na *heap*, área de memória reservada para alocação de objetos dinâmicos, assim o coletor de lixo da JVM gerencia uma única região de memória.

Vários tipos de algoritmos de coleta de lixo foram implementados nas várias versões da JVM, neste trabalho será feito um estudo sobre os algoritmos existentes no Hotspot JVM versão 1.6 que, no processo de evolução, é considerado o mais eficiente. Hotspot é o nome dado pela Sun para sua implementação da máquina virtual, pois qualquer pessoa pode criar uma implementação para JVM, desde que siga corretamente as especificações disponibilizadas pela Sun.

(GOENZ, 2003) A partir da versão 1.2 da JVM, a técnica de coleta em gerações (*gerational garbage collection*) foi implementada combinada com a técnica de cópia (*copying collector*) e *mark-compact*, para tentar unir o que há de melhor em cada um destes algoritmos. O algoritmo de cópia funciona muito bem quando uma grande quantidade de objetos se torna lixo e tem o desempenho reduzido quando muitos objetos

têm o tempo de vida muito longo, pois serão copiados sempre. Ao contrário do algoritmo de cópia, o *mark-compact* funciona muito bem com objetos que tem o tempo de vida muito longo (por serem copiados apenas uma vez), e não muito bem com objetos que tem tempo de vida muito curto.

A natureza geracional do sistema de memória da JVM provê maior flexibilidade para o uso de algoritmos específicos de coleta de lixo de acordo com o ambiente e o aplicativo em execução. (Microsystems, 2006) A JVM suporta diversos algoritmos diferentes de coleta de lixo de acordo com a necessidade do aplicativo, pensando no tempo de pausa e na taxa de *throughput*.

Os coletores da JVM são totalmente exatos, diferente de muitos outros algoritmos que são conservativos. Algoritmos conservativos (*conservative garbage collector)*, como já descritos neste trabalho, não sabem exatamente onde estão localizadas as referências para os objetos e assim não realizam a coleta de forma correta, deixando de coletar vários objetos.

Os algoritmos exatos podem dar algumas garantias que os algoritmos não exatos não podem, como por exemplo:

- Todos os objetos inacessíveis são reciclados com certeza absoluta;
- Todos os objetos podem ser realocados, permitindo a compactação dos objetos na memória, o que elimina a fragmentação da memória e aumenta a localidade de memória (principio de localidade de referência);

A memória da JVM é dividida em três gerações: jovem, antiga e permanente. A geração jovem é onde a maioria dos objetos é alocada e ocupa um espaço menor na memória. A geração antiga contém os objetos que sobreviveram a um determinado número de coletas na geração jovem; alguns objetos grandes podem ser alocados diretamente na geração antiga, por ter um tamanho bem maior em proporção ao tamanho da geração jovem. A geração permanente é um espaço de memória alocada por processos não relacionados à criação de objetos, como por exemplo, o carregamento de um classe, a área de métodos de classes (código compilado), classes geradas dinamicamente e objetos nativos ao java (JNI).

Existem coletores na JVM, tanto para geração jovem quanto para geração antiga, que executam em série (para ambiente mono-processado) ou em paralelo (para ambientes multi-processados), tais algoritmos estão descritos a seguir.

3.1. Geração Jovem

A geração jovem é dividida em três áreas, um espaço chamado Éden e mais dois outros menores chamados de espaço dos sobreviventes, como mostra a figura 6. A maioria dos objetos é alocada no Éden enquanto que alguns objetos muito grandes são alocados diretamente na geração antiga. No espaço sobrevivente estão os objetos que sobreviveram a pelo menos uma coleta na geração jovem. Sempre que um objeto sobrevive a uma coleta na geração jovem sua idade aumenta, com isso as chances deste objeto se tornar lixo também aumentam. Mas a partir de uma determinada idade o objeto é promovido para a geração antiga, pois como ele ainda não se tornou lixo é bem provável que ele continue vivo por um bom tempo.

Os dois espaços sobreviventes são usados para a realização do *copying collection*, um deles é nomeado de espaço origem e o outro de espaço destino. Estes dois espaços alternam a função de origem e destino e sempre um deles está vazio.

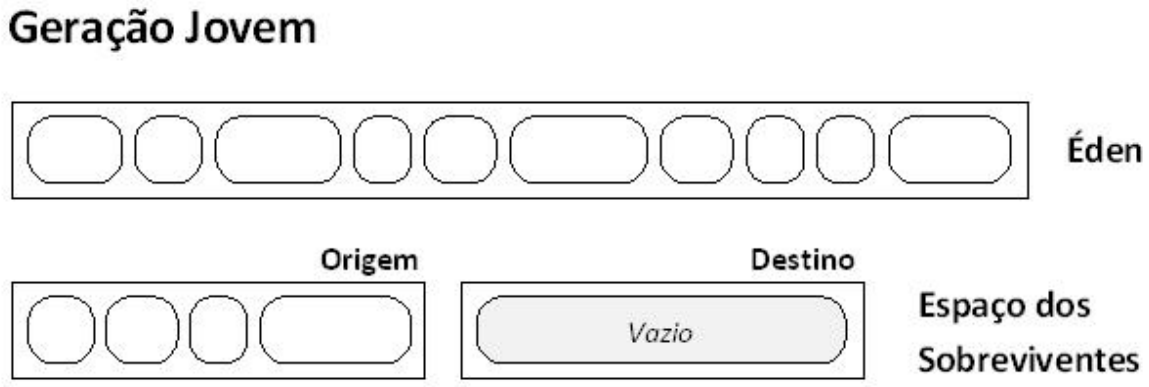

Figura 7 – Áreas de memória da geração jovem

Na execução do *copying collection*, o algoritmo utiliza o Éden e os dois espaços de sobreviventes, sendo o Éden e um dos sobreviventes os espaços de origem, enquanto o outro espaço sobrevivente e a geração antiga são considerados os espaços de destino.

Quando o espaço das gerações antiga e permanente se acaba, o coletor é ativado para realizar a coleta de lixo completa, conhecida como coleta maior. Nesse processo todas as gerações são coletadas. Normalmente a geração jovem é coletada primeiro, mas se a geração antiga estiver cheia pode ser necessário coletar a geração antiga primeiro, pois pode ocorrer de alguns objetos tenham que ser promovido e sendo assim copiados para a geração antiga.

Com o algoritmo *copying collection* sendo aplicado na geração jovem, então sempre existe um grande bloco contíguo de memória disponível para alocar novos objetos. Para isso o coletor armazena uma referência para o final do último bloco alocado. A alocação a partir deste bloco é eficiente, pois para se realizar uma nova alocação tudo que precisa ser feito é checar se o novo objeto cabe na área livre disponível para alocação; se couber, atualiza a referência para o fim do bloco contiguo de memória livre e inicializa o objeto.

Um cuidado adicional deve ser tomado com aplicativos *multithread*, já que existe risco de múltiplas alocações serem feitas simultaneamente. Assim as operações de alocação precisam ser seguras, com o cuidado de que uma alocação seja feita por vez. Se um *lock* global fosse utilizado para este fim, a alocação nessa geração poderia se tornar um gargalo do sistema, reduzindo o desempenho do aplicativo. Para resolver este problema, o Hotspot JVM utiliza uma técnica chamada *Thread-Local Allocation Buffers* (TLABs, buffer de alocação local à thread), que consiste em dar a cada thread um buffer (uma pequena porção da geração) para alocação. Esta técnica aumenta a taxa de *throughput* de alocação em aplicações *multithread* sem que seja necessário um *lock* global para realizar as alocações de forma segura. O único problema é quando a TLAB de uma *thread* fica cheia: neste caso, para se adquirir mais um buffer, as threads são obrigadas a sincronizar.

3.1.1. Coleta em Série

Como ilustrado na figura 7, o algoritmo executado na geração jovem funciona da seguinte forma: os objetos alcançáveis que estão no Éden são copiados para o espaço de sobreviventes destino, com exceção dos objetos que não cabem neste espaço, tais objetos são copiados diretamente para a geração antiga.

Os objetos que ainda são alcançáveis e estão no espaço de sobreviventes origem e que ainda são considerados jovens também são copiados para o espaço de sobreviventes destino, enquanto que os objetos considerados antigos são promovidos e copiados para a geração antiga.

Se o espaço de sobreviventes destino estiver cheio e ainda existir objetos alcançáveis no Éden e no espaço de sobreviventes origem, os objetos restantes serão copiados diretamente para a geração antiga independentemente da quantidade de coletas às quais eles sobreviveram. Isso para impedir que a geração jovem fique em um estado inconsistente após a coleta.

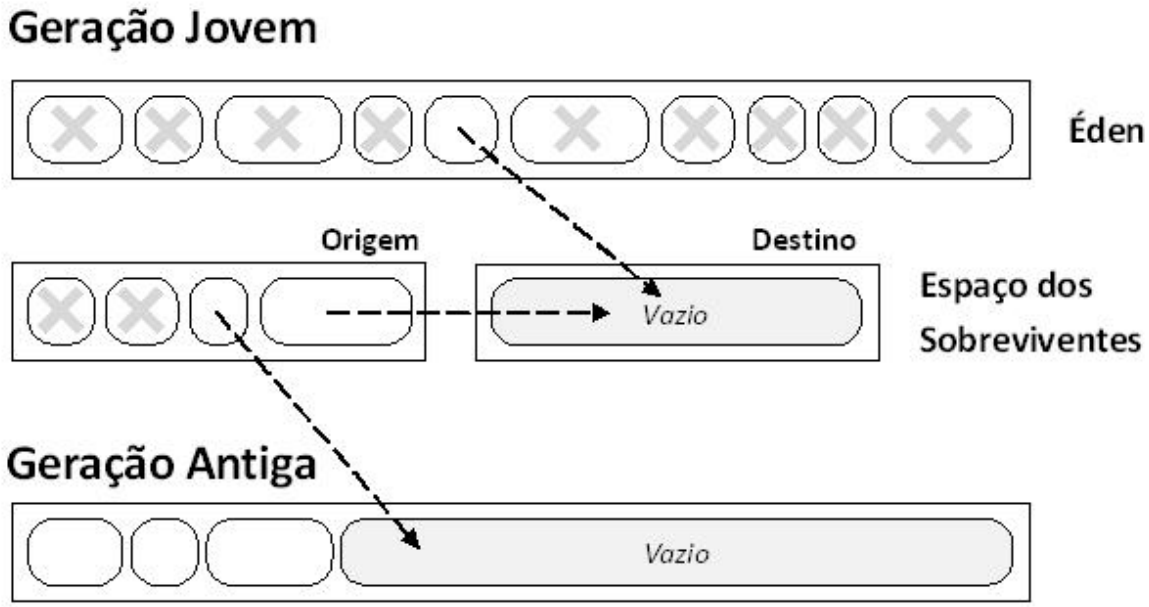

Figura 8 – Coleta na geração jovem (antes)

Os objetos restantes no Éden e no sobrevivente origem são considerados lixo e não precisam ser examinados (esses objetos estão marcados com um X na figura, mas o algoritmo não marca nem examina estes objetos).

No final da execução do algoritmo, tanto o Éden quanto o espaço de sobrevivente de origem ficam vazios e somente o espaço de sobreviventes de destino contém os objetos alcançáveis. A partir desse ponto, os espaços de sobreviventes trocam de função (quem era destino passa a ser origem, e vice versa).

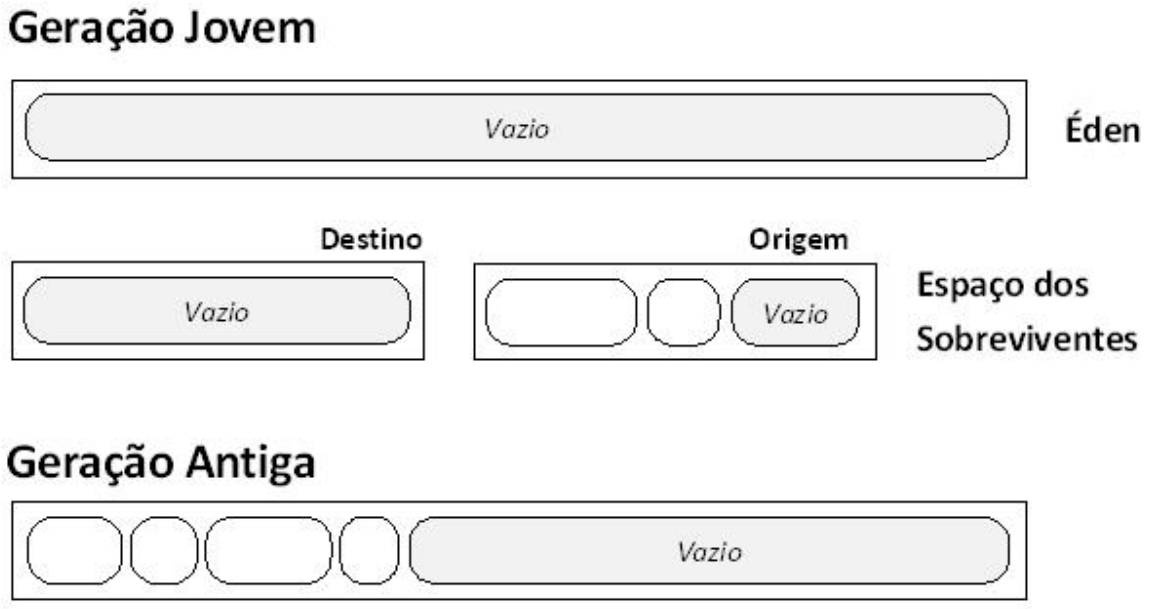

Figura 9 – Coleta na geração jovem (depois)

3.1.2. Coleta em Paralelo

A Hotspot JVM tem um algoritmo de coleta de lixo em paralelo para ambientes multi-processados. Este tenta evitar que o coletor execute em um processador enquanto os outros ficam inativos, com isso não há desperdício de recursos do sistema. Esse algoritmo é uma versão paralelizada do anterior, ainda necessita que o aplicativo pare sua execução momentaneamente e também utiliza o *copying collector*, mas executa o coletor em paralelo. O coletor simplesmente divide a tarefa de copiar os objetos do espaço origem para o espaço destino, entre os processadores disponíveis. Isso reduz o tempo de execução do algoritmo e conseqüentemente o tempo de espera do aplicativo.

3.2. Geração Antiga

A geração antiga é composta por um único espaço contiguo de memória, geralmente bem maior que a geração jovem. Neste espaço estão os objetos que sobreviveram a várias coletas menores, ou seja, os objetos copiados várias vezes de uma área de sobreviventes para a outra. O algoritmo usado na geração jovem é quem decide quando promover um objeto para essa geração. Os objetos muito grandes em proporção da geração jovem são alocados diretamente nessa geração.

O processo de coleta de lixo na geração antiga é chamado de coleta maior (ou coleta completa). As coletas menores gradualmente enchem essa região, quando esta geração fica cheia, o processo de coleta é executado em todas as gerações a partir da geração jovem.

A coleta na geração antiga pode ocorrer antes se por acaso uma coleta menor não puder ser executado por falta de espaço, caso o Éden e os sobreviventes estejam cheios com objetos alcançáveis.

Para a geração antiga existem três tipos de coletores disponíveis no Hotspot JVM, são eles: em série, em paralelo e o concorrente. O coletor em série é o utilizado pela JVM por padrão, mas dependendo da natureza do aplicativo, uma das outras opções podem ser selecionadas.

3.2.1. Coleta em Série

O algoritmo utilizado na coleta em série é o *mark-compact*. O algoritmo é classificado como em série por ser uma implementação para ambiente mono-processado, além disso, necessita parar a execução do aplicativo momentaneamente para realizar a coleta.

O algoritmo, como já foi visto, é dividido em três fases. Na primeira fase faz-se a marcação dos objetos que são alcançáveis a partir do conjunto raiz, usando a propriedade da transitividade. Na segunda fase é realizada uma varredura na geração, verificando e identificando quais objetos são lixos de memória e podem ser reciclados. Na última fase o coletor realiza a compactação, arrastando todos os objetos alcançáveis para o inicio do espaço da geração antiga. Isso mantém um único espaço contiguo de memória para alocações futuras.

Esse algoritmo é o recomendado para a maioria dos aplicativos desktop.

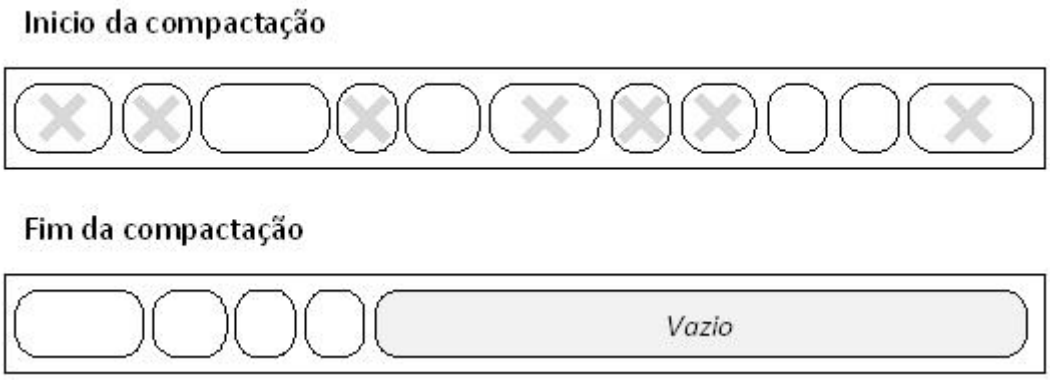

Figura 10 – Compactação da geração antiga

3.2.2. Coleta em Paralelo

O algoritmo paralelo para a geração antiga é uma variação do *mark-sweep* mesclado com o *copying collection*. Esse algoritmo também necessita que o aplicativo pare sua execução momentaneamente. O processo de coleta é divido em várias *threads*, onde cada uma pode ser executado em um processador (caso a JVM esteja sendo executado em um ambiente multiprocessado).

O processo de coleta é dividido em três fases. Inicialmente a geração é dividida em regiões de tamanho fixo, onde são armazenadas informações sobre os objetos que se encontram nela. Na primeira fase, a de marcação (fase herdada do *mark-sweep*), os objetos do conjunto raiz são divididos entre as *threads* do coletor e com isso todos os objetos ainda vivos são marcados em paralelo. Sempre que um objeto é identificado com vivo, os dados da região onde este objeto se encontra são atualizados com as informações sobre o tamanho e a localização deste objeto.

A segunda fase opera sobre as regiões, não necessitando, portanto que sejam feitas quaisquer processamentos com os objetos. Devido à compactação realizada pelas coletas

anteriores, existe uma grande tendência que as porções do lado esquerdo da geração estejam densas, contendo muitos objetos vivos, como ilustrado na figura 11. A quantidade de espaço que poderia ser recuperado dessas regiões densas não vale o custo de compactá-las. Então a primeira coisa a ser feita nesta fase é examinar a densidade das regiões, iniciando a verificação da esquerda, onde as regiões são mais densas, até alcançar um ponto onde valha a pena executar a compactação, ou seja, onde a relação espaço que pode ser recuperado e custo de compactação seja o maior possível. As regiões à esquerda deste ponto são marcadas com um prefixo denso e nenhum objeto será movido nessas regiões. Todas as regiões à direita deste ponto serão compactadas, eliminando os espaços dos objetos inalcançáveis. Nessa fase é calculada e armazenada a nova localização do primeiro byte dos dados vivos para cada região.

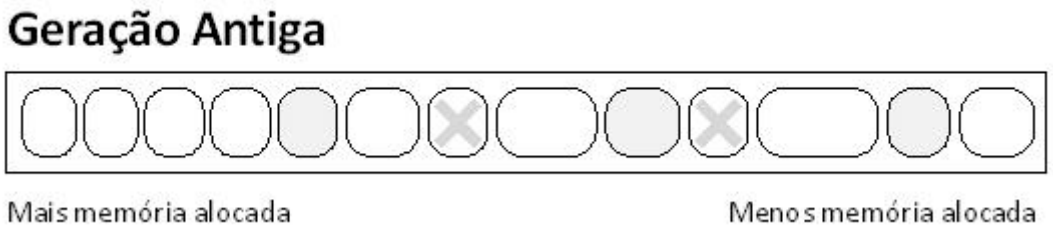

Figura 11 – Densidade de objetos

A segunda fase é executada serialmente. De acordo com (Microsystems, 2006), a paralelização desta fase é possível, mas não causa tanto impacto no desempenho quanto a paralelização das fases de marcação e de compactação.

Na terceira fase é realizada a compactação dos objetos nas regiões definidas pela fase anterior. As várias threads do coletor usam os dados criados pela segunda fase para identificar quais regiões necessitam ser compactadas. Com esses dados cada thread pode copiar independentemente os dados nas regiões. Ao final dessa fase, uma extremidade dessa geração terá um bloco denso de dados e a outra extremidade terá um único bloco de espaço livre.

Esse algoritmo é recomendado para aplicativos que executam em máquinas com mais de um processador, reduzindo o tempo de espera do aplicativo. Dependendo da quantidade de *threads* em ambientes monoprocessados pode haver muita concorrência entre as *threads*, isso pode acarretar em perda de desempenho.

3.2.3. Coleta Concorrente

Para muitos aplicativos, a taxa de *throughput* não é tão importante quanto o tempo de resposta. Em geral, a geração jovem não é a responsável pelas longas paradas do coletor: as coletas na geração antiga, apesar de serem executadas poucas vezes, são as responsáveis, principalmente quando o aplicativo possui a necessidade de utilizar uma *heap* muito grande. Para reduzir o tempo de latência, o Hotspot JVM incluiu um coletor chamado *concurrent mark-sweep* (CMS), também conhecido como coletor de baixa latência.

Assim como o *mark-sweep*, o CMS é divido em fases. Na fase inicial uma pequena pausa é feita para identificar o conjunto inicial de objetos alcançáveis a partir do código do aplicativo (o conjunto raiz).

Na segunda fase, o coletor marca todos os objetos vivos que são transitivamente alcançáveis a partir do conjunto raiz. Esse processo é realizado de forma concorrente, pois enquanto o aplicativo está executando e alterando os campos dos objetos, o coletor está realizando a marcação dos objetos alcançáveis.

Não há a garantia de que todos os objetos estejam marcados no fim da segunda fase. Por isso, na terceira fase, conhecida como fase de remarcação, o coletor faz uma pequena pausa no aplicativo para finalizar o processo de marcação, revisitando alguns objetos que foram modificados durante a segunda fase. Esse processo é bem mais eficiente que o realizado na fase inicial, pois múltiplas *threads* são empregadas para realizar o processo de remarcação em paralelo.

No final da fase de remarcação, com certeza todos os objetos vivos estarão marcados. Por fim é executada a última fase do coletor, chamada de fase de varredura simultânea. Nessa fase, enquanto o aplicativo está em execução, o coletor identifica e recicla todos os objetos não marcados.

Figura 12 – Fase de verificação do CMS

Esse algoritmo não realiza a compactação dos objetos vivos, com isso a memória sempre fica fragmentada após a execução do coletor, como mostra a figura 12. Por não realizar a compactação, esse algoritmo economiza tempo, mas deixa o espaço livre de forma não contígua. Por causar da fragmentação, o processo de alocação deve armazenar em uma lista todos os espaços livres disponíveis, de forma que seja possível posteriormente realizar alocações.

Capitulo 4

Novas Heurísticas para Coleta de Lixo na JVM

Apresentaremos ao longo deste capítulo as heurísticas propostas para realizar a coleta de lixo na máquina virtual Java. Antes, porém, apresentaremos alguns detalhes da implementação dos distintos coletores existentes na JVM, uma vez que estas informações serão importantes para o entendimento das modificações que estamos propondo.

4.1. Implementação das Técnicas de Coleta de Lixo na JVM

A implementação da JVM introduz o conceito de espaço, uma abstração para as "unidades de armazenamento", ou seja, para pedaços da *heap* que compõe toda ou parte de uma geração. Um objeto da classe espaço inclui métodos que mantém controle das regiões livres e utilizadas, realizam processamento sobre objetos e blocos livres, dentre outras funcionalidades.

Uma hierarquia de classes é utilizada pela JVM para definir os espaços, agrupando-os em grupos. Um espaço pertence a um determinado grupo caso tenha as mesmas características que os outros espaços que compõe aquele mesmo grupo. A hierarquia segue o padrão ilustrado na figura 13.

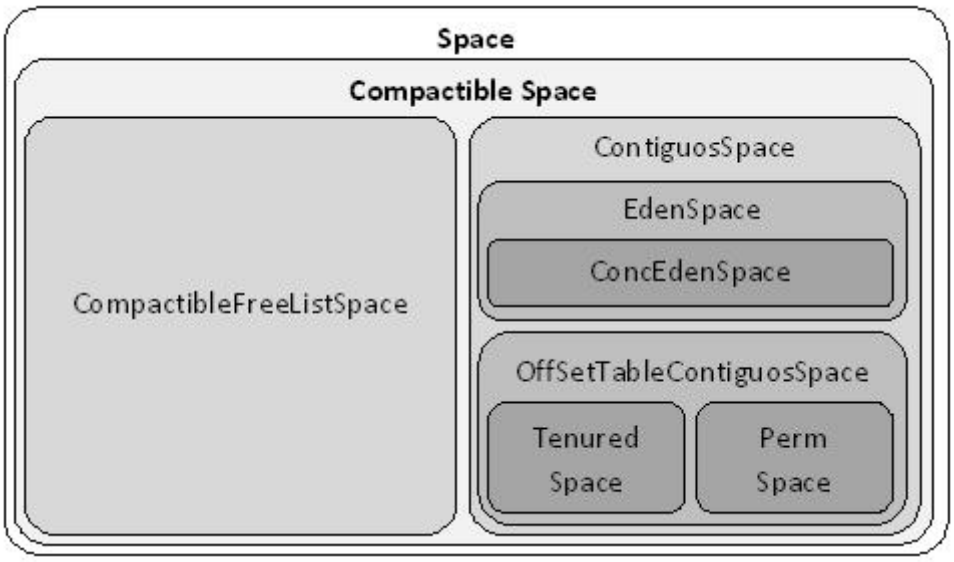

Figura 13 – Hierarquia de classes para definir espaços

- ***Space***: é a classe pai de todos os espaços da JVM. Trata-se de uma classe abstrata que rotula todos os espaços em um mesmo grupo. O Space descreve uma *heap* que dá suporte a alocação de objetos, cálculo do tamanho da *heap* e suporte à coleta de lixo.
- ***CompactibleSpace***: é um espaço que suporta compactação da *heap*. A alteração das referências a objetos que serão movidos é feita facilmente neste espaço. Esse espaço tem todas as características do espaço pai (*Space).*
- ***CompactibleFreeListSpace***: esse tipo de espaço é o utilizado pelo *Concurrent Mark Sweep*. Após a execução deste coletor, a memória fica fragmentada. Para que novas alocações possam ser realizar nesse espaço, uma lista de espaços livres deve existir. Este espaço tem suporte para que o coletor funcione, mantendo uma lista de espaços disponíveis. Esta lista é criada sempre após a execução do coletor. Esse espaço também suporta compactação, ou seja, se for necessário, todo o espaço pode ser compactado para manter um único bloco de espaço livre contíguo.
- ***ContiguousSpace***: esse tipo de espaço é compactável e todo o seu espaço livre é contíguo, diferentemente da classe descrita anteriormente. Assim, nesta classe não existe a figura da lista de espaços livres. Por ter um espaço livre contíguo, esse espaço suporta alocações rápidas de novos blocos de memória: basta apenas verificar se o bloco requisitado cabe no espaço disponível; se couber, apenas incrementa-se o topo da *heap*.
- ***EdenSpace***: esse é o espaço padrão da JVM para a geração jovem. Suporta alocações rápidas e compactação, além de que o espaço livre está sempre contíguo.
- ***ConcEdenSpace***: esse espaço é idêntico ao anterior, com a diferença suportar alocações concorrentes.
- ***OffsetTableContigSpace***: é um espaço contíguo dividido em blocos. Um vetor armazena os *offsets* de cada bloco alocado, o que proporciona acesso rápido ao inicio de um bloco, mecanismo este particularmente útil em espaços relativamente grandes. Essa classe serve de abstração para as gerações mais antigas (gerações antiga e permanente).
- ***TenuredSpace***: esse espaço é usado na geração antiga, e tem todas as características do seu espaço pai (*OffsetTableContigSpace).*
- ***ContigPermSpace***: esse espaço é semelhante ao anterior, mas usado na geração permanente.

Uma geração modela uma área da *heap* que contém objetos de mesma idade. Uma geração pode ser composta de um ou vários espaços. Também existe uma hierarquia no modelo de classes das gerações, conforme ilustrado na figura 14.

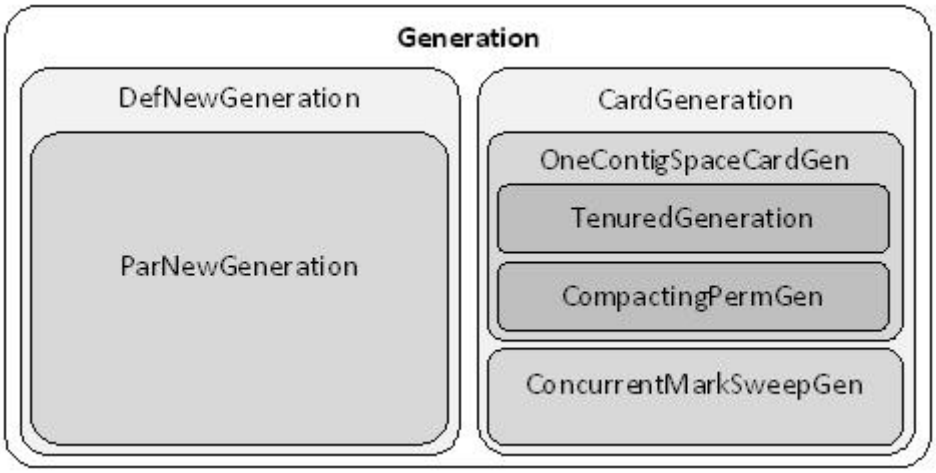

Figura 14 – Hierarquia de classes para definir gerações

- **Generation**: classe abstrata que representa todas as gerações existentes na JVM. É responsável por gerar e armazenar estatísticas sobre os espaços por ela gerenciados, assim como dados sobre o coletor que age sobre esta geração. A classe também manipula os objetos que estão presentes em seus espaços, podendo alterar a localização e as referências de um objeto. Por fim, é responsável por decidir em qual espaço novas alocações podem ser realizadas e também por preparar os espaços para uma execução do coletor de lixo.
- ***DefNewGeneration***: é a classe que representa a geração jovem e usa o coletor de cópia;
- ***ParNewGeneration***: semelhante ao anterior, mas é coletada em paralelo por várias *threads*;
- ***CardGeneration***: uma classe abstrata com manipuladores de vetor de *offset*. Os espaços que podem ser utilizados nessa geração devem suportar a divisão em blocos.
- ***OneContigSpaceCardGeneration***: essa classe abstrata contém apenas um espaço contíguo e suporta marcação de *card* (neste caso, o espaço deve suportar divisões em blocos, sendo cada bloco chamado de *card*). O coletor suportado para essa geração é o *Mark Compact*.
- ***TenuredGeneration***: geração que armazena os objetos mais antigos.
- ***CompactingPermGenGen***: geração que armazena as classes, métodos, símbolos, etc.

- ***ConcurrentMarkSweepGeneration***: uma geração que suporta o algoritmo de coleta *Concurrent Mark Sweep*.

A JVM trabalha com três gerações: geração jovem, antiga e permanente. Como já foi visto, na geração jovem ficam os objetos recém alocados. Conforme o coletor escolhido, a geração jovem é definida pela classe *DefNewGeneration* ou pela classe *ParNewGeneration*. A geração jovem é composta de três espaços: o Éden, representado pelas classes *EdenSpace* ou *ConcEdenSpace*, e dois espaços de sobreviventes, o de origem e o de destino, ambos representados pela classe *ContiguousSpace*.

Na geração antiga ficam os objetos que sobreviveram a um determinado número de coletas. Pode ser representada tanto pela classe *TenuredGeneration* quanto pela classe *ConcurrentMarkSweepGeneration*. A geração antiga contém apenas um espaço do tipo *TenuredSpace* ou *CompactibleFreeListSpace*, dependendo do coletor escolhido para essa geração.

Na geração permanente ficam alocadas as classes, métodos e símbolos, dentre outros. Pode ser representada pela classe *CompactingPermGenGem* ou pela classe *ConcurrentMarkSweepGeneration*. Tem apenas um espaço, do tipo *ContigPermSpace*.

As combinações das gerações atualmente suportadas pela JVM para as gerações jovem, antiga e permanente, são respectivamente:

- DefNewGeneration + TenuredGeneration + PermGeneration;
- DefNewGeneration + ConcurrentMarkSweepGeneration + ConcurrentMarkSweepPermGen;
- ParNewGeneration + TenuredGeneration + PermGeneration;
- ParNewGeneration + ConcurrentMarkSweepGeneration + ConcurrentMarkSweepPermGen;

A JVM trabalha com apenas uma *heap* para cada aplicativo em execução. Para isso, cada universo de execução contém apenas um objeto que define a *heap*. Essa *heap* pode ou não englobar gerações; caso tenha gerações, cada geração tem definido seu respectivo coletor. Também existe uma hierarquia de classes para os tipos de *heap* existentes na JVM, como ilustrado na figura 15.

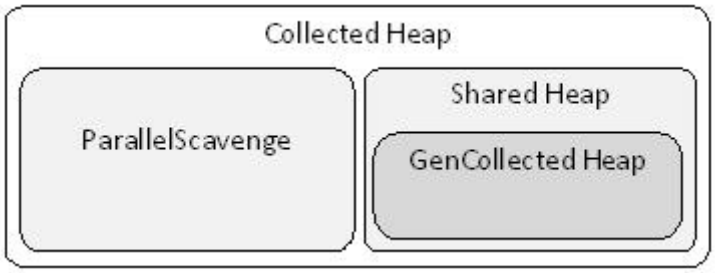

Figura 15 – Hierarquia de classes para definir a *heap*

- ***CollectedHeap***: é uma classe abstrata que define todas as funções que uma *heap* deve implementar, além de conter a infra-estrutura padrão para todos os modelos de *heap*.
- ***SharedHeap***: é uma classe com as características da classe pai (*CollectedHeap*). Apenas uma única instância de *SharedHeap* pode existir em um aplicativo.
- ***GenCollectedHeap***: essa classe define uma *heap* do tipo *SharedHeap*, mas com o modelo geracional, englobando todas as gerações em uma única *heap*.
- ***ParallelScavengeHeap***: esse modelo de *heap* utiliza gerações, mas não suporta nenhum dos tipos de geração definidos anteriormente: ela tem suas próprias definições de geração e espaço, para ter o suporte necessário para o coletor *ParallelScavenge*.

A *CollectedHeap* utiliza os serviços de um gerenciador de eventos da JVM. Esse gerenciador é responsável por enviar sinais para toda a máquina virtual. Os sinais informam a máquina sobre o estado atual do sistema. Por exemplo, quando um coletor inicia a execução, o gerenciador de eventos é avisado sobre qual região está sendo coletada; o gerenciador de eventos é então responsável por avisar a todas as *threads* que reservaram espaços naquela região de que ela estará bloqueada até o fim da coleta. Todo esse processo não é simples, pois qualquer mudança de estado da máquina virtual deve ser avisada ao gerenciador de eventos para que não deixe a máquina em um estado inconsistente.

4.1 DefNewGeneration

A geração jovem é a responsável por manter os objetos recém alocados, até que estes sobrevivam a um determinado número de coletas ou sejam reciclados. Essa geração gera e armazena dados estatísticos sobre todo o processo de alocação e coleta de lixo. Esses dados são analisados pela própria máquina para tomar decisões quanto a realização de coletas, promoção de objetos, dentre outras operações. Alguns destes dados compõem as informações geradas e analisadas pela *CollectedHeap*.

Vale lembrar que a geração jovem é bem menor que a geração antiga, pelo fato de que os objetos ficarão pouco tempo nela. A maioria dos objetos tende a morrer muito cedo e os sobreviventes são copiados para a geração antiga. Os objetos contidos na geração antiga tendem a continuar vivos por um grande período de tempo, talvez até o fim da execução do aplicativo.

Quanto à alocação, todo o processo é gerenciado por uma classe que define políticas de coleta. Esta classe é conhecida como gerente de alocação. O processo de alocação passa por várias etapas antes que um novo objeto seja alocado. Inicialmente faz-se uma requisição de espaço para uma geração, no caso para a geração jovem. A geração jovem repassa a requisição para o gerente de alocação, que verifica se existe espaço disponível para atender a requisição. Duas situações distintas podem levar o gerente a não atender a requisição de imediato:

a) O espaço requisitado é muito grande para ser alocado na geração jovem. Neste caso o gerente repassa a requisição para a geração antiga, que como descrevemos, é bem maior que a geração jovem.
b) Os espaços gerenciados pela geração estão cheios, neste caso o coletor é chamado para realizar a limpeza dos espaços e/ou promoção dos objetos sobreviventes.

O gerente de alocação recebe alguns dos sinais do gerenciador de eventos da JVM. Um sinal importante é o *jni critical section*, que indica se existe alguma *thread* acessando uma região crítica na geração jovem. O coletor não pode ser iniciado se por acaso alguma *thread* esteja acessando sua região crítica nessa geração. Para evitar problemas, antes de iniciar o coletor, o gerente de alocação verifica a existência deste sinal no gerenciador de eventos. Caso existam *threads* acessando sua região crítica, o gerente de alocação aguarda por um novo sinal do gerenciador de eventos que indique que a *thread* terminou seu acesso à região crítica.

Quando o gerente de alocação decide executar o coletor, ele inicia uma operação de coleta através da criação de um novo objeto do tipo *VM_GC_Operation*. Existem vários tipos de operações de coleta de lixo, e essas operações são definidas pelas seguintes classes:

- ***VM_GC_Operation***: essa classe implementa todos os métodos comuns a todas as classes na hierarquia. Ela também previne múltiplas execuções simultâneas de coleta de lixo e gerencia o *lock* sobre a *heap*;
- ***VM_GC_HeapInspection***: responsável por imprimir, caso seja especificado na

execução do aplicativo, um histograma de classes, ou seja, imprime os dados estatísticos obtidos durante a execução;

- ***VM_GenCollectForAllocation***: esta operação é executada quando ocorre uma falha de alocação. Neste caso, quando o gerente de alocação detecta que não existe espaço disponível para alocação na geração, o coletor é executado fazendo uma chamada para essa operação. O tipo de coletor a ser executado é determinado pela geração que invocou esta operação.
- ***VM_GenCollectFull***: essa operação executa uma coleta completa da *heap*. Esta coleta pode ser realizada de duas formas: iniciando da geração jovem ou iniciando da geração antiga.
- ***VM_GenCollectFullConcurrent***: operação idêntica a classe anterior, mas para coletor concorrente.
- ***VM_ParallelGCFailedAllocation***: esta operação é executada quando ocorre uma falha de alocação em algum espaço em uma geração que utilize o coletor paralelo.
- ***VM_ParallelGCFailedPermanentAllocation***: operação idêntica a da classe anterior, mas para a geração permanente.
- ***VM_ParallelGCSystemGC***: essa operação executa uma coleta completa da *heap*. Esta operação de coleta é usada quando o coletor paralelo é o escolhido, podendo ser realizada executada de duas formas: iniciando da geração jovem ou iniciando da geração antiga.

O gerente de alocação executa na geração jovem uma operação do tipo *VM_ GenCollectForAllocation* com o objetivo de executar uma coleta nessa geração. Essa operação envia alguns sinais para o gerenciador de eventos que fazem com que toda a execução da máquina pare momentaneamente enquanto a coleta é realizada.

Nessa geração o coletor marca os objetos que estão vivos no Éden e no espaço de sobreviventes origem, copiando-os para o espaço de sobrevivente destino ou promovendo-os para a geração antiga. O processo de marcação parte do conjunto raiz e verifica quais são os objetos alcançáveis. Um primeiro problema, no entanto, é justamente definir o conjunto raiz. A JVM utiliza classes que percorrem todos os componentes alcançáveis, como classes, métodos, objetos, vetores etc. Essas classes são chamadas de fechos.

Para que um fecho possa percorrer todos os componentes vivos, estes devem possuir suporte à essa tarefa, por isso todos os componentes foram implementados de acordo com o padrão de projeto *Visitor* (ALEXANDRESCU, 2001).

O padrão *Visitor* é uma maneira de separar um algoritmo da estrutura de um objeto. O objetivo do padrão *Visitor* é representar uma operação a ser realizada nos elementos de um objeto. O *Visitor* permite definir uma nova operação sem ter de alterar as classes nos quais a operação atua. Isso facilita a tarefa de percorrer os objetos na JVM, pois as referências contidas em cada objeto são conhecidas apenas em tempo de execução. O *Visitor* realiza a operação desejada no objeto e repassa a chamada da operação para os filhos.

Um fecho é responsável por realizar a marcação dos objetos vivos. O processo inicia-se na geração permanente, percorrendo todos os componentes dessa geração. Em cada componente o fecho percorre as referências para os objetos alocados em outras gerações. Cada objeto visitado pelo fecho é marcado. No final do processo, apenas os objetos vivos estarão marcados.

Depois de realizar a marcação dos objetos vivos no Éden, o coletor os copia para o espaço de sobrevivente destino. Caso não caibam no espaço de destino, os objeto são promovido, ou seja, copiados para a geração antiga. Os objetos ainda jovens marcados no espaço de sobrevivente origem são copiados para o espaço de sobreviventes destino enquanto os objetos mais antigos são promovidos. No final do processo, tanto o Éden quanto o espaço de sobrevivente origem estarão vazios, enquanto o espaço de sobrevivente destino estará cheio. Neste ponto, os espaços de sobreviventes trocam os papéis; o espaço de origem passa a ser destino e vice-versa.

Podem ocorrer falhas durante a promoção de alguns objetos caso a geração antiga esteja cheia. Esse problema pode ocorrer no meio do processo cópia do coletor, ou seja, alguns objetos serão copiados enquanto outros não. Para resolver este problema, os objetos já copiados são considerados lixo na geração antiga. No espaço livre que foi gerado por esses objetos são alocados novos objetos temporários com a finalidade de causar uma nova coleta. Essa nova coleta será completa, sendo iniciada pela geração antiga, tendo por objetivo gerar o espaço necessário para a realização da promoção dos objetos que não puderam ser copiados por falta de espaço.

Quando o coletor não consegue liberar espaço em nenhuma das gerações, uma exceção é lançada, indicando que não há memória disponível para novas alocações. Assim, o aplicativo em execução é forçosamente encerrado.

4.3 Heurísticas Propostas

A execução do coletor é considerada um custo adicional para manutenção da memória. Para reduzir este custo deve-se reduzir o tempo e/ou número de execuções do coletor, já que quanto menor for o tempo gasto com coletas, menor será o tempo total de execução do aplicativo.

Uma forma de reduzir o custo de execução de um coletor é atacando os pontos em que o coletor perde mais tempo executando. No caso do coletor da geração jovem, o maior custo é realizar a cópia dos objetos vivos.

Outra forma de reduzir o tempo total de execução do aplicativo é diminuir a quantidade de coletas. Para evitar a execução do coletor deve-se eliminar as falhas de alocação e uma das formas de se fazer isto é aumentar o tamanho da *heap*. A JVM trabalha com uma *heap* de tamanho constante. A única forma de se alterar o tamanho da *heap* é através da passagem de parâmetros durante a criação da JVM, no momento em que o aplicativo é chamado.

4.3.1. Redução do Custo do Coletor

O coletor da geração jovem é o mais executado durante a execução da máquina virtual. Reduzir o custo do coletor é o mesmo que reduzir o tempo de pausa do aplicativo e com isso o seu tempo total de execução.

Uma das formas de se reduzir o custo da geração jovem é diminuindo-se a quantidade de cópia realizada pelo coletor. Por exemplo, simplesmente poderíamos marcar os objetos vivos e sempre que uma requisição de memória fosse feita, retornar-se-ia o espaço de algum objeto não marcado, pois tal objeto é considerado lixo de memória. Para realizar esse processo, o gerente de alocação deve ser alterado, de forma a suportar esse tipo de operação. O espaço que realiza as alocações também deveria ser alterado.

A idéia inicial deste trabalho era de alterar o coletor da geração jovem para tentar reduzir o tempo de espera, substituindo o coletor de cópia pelo coletor *Mark Lazy-Sweep*.

De acordo com (LINS, 1996), o *Mark Lazy-Sweep* consiste em marcar os objetos vivos assim como no algoritmo de cópia. Na alocação de novos objetos, este algoritmo procura na *heap* os objetos não marcados para poder alocar o novo objeto no lugar do objeto não marcado encontrado. Esse algoritmo reduz o tempo de pausa do coletor, mas transfere o custo do coletor para a fase de alocação. O coletor é praticamente

implementado no processo de alocação, como visto no algoritmo de Hughes (LINS, 1996), descrito a seguir em alto nível de abstração:

```
função aloca() : endereço_memoria
   enquanto (objeto < topo_heap) faça
      se marcado(objeto) então
         desmarca(objeto);
         objeto = objeto + tamanho(objeto);
      senão
         result = objeto;
         objeto = objeto + tamanho(objeto);
         retorna result;
      fim-se
   fim-enquanto
   marca_heap();
   objeto = inicio_heap();
   enquanto (objeto < topo_heap) faça
      se marcado(objeto) então
         desmarca(objeto);
         objeto = objeto + tamanho(objeto);
      senão
         result = objeto;
         objeto = objeto + tamanho(objeto);
         retorna result;
      fim-se
   fim-enquanto
   exceção("Sem memória");
fim-função
```

- Continua a procura por objetos desmarcados a partir do ponto anterior (última alocação)

- Se chegar neste ponto a *heap* está cheia, então remarca os objetos e reinicia a procura

- Se chegar neste ponto a *heap* está cheia, e não tem lixo na memória, ou seja, acabou a memória

Uma variação deste algoritmo consiste em realizar a marcação e logo em seguida criar uma lista com os espaços disponíveis, e no processo de alocação verificar na lista se existe espaço disponível para atender a requisição.

Para implementar este coletor na JVM, foi necessário alterar o coletor da classe *DefNewGeneration*, bem como o gerente de alocação dessa geração. No *DefNewGeneration* o processo de marcação não foi alterado, e ao processo de cópia foi adicionado o gerador da lista de espaços disponíveis. O algoritmo em alto nível que representa o processo de alocação realizado pelo gerente de alocação atualmente existente na JVM é o seguinte:

```
função faz_alocação(tamanho) : endereço_memória
   gera_estatisticas();
   enquanto (verdade) faça
      objeto = tenta_alocar(geração_jovem_eden, tamanho);
      se não_nulo(objeto)
         então retorna objeto;
      fim-se
      adquiri_lock_heap();
      se muito_grande(objeto) então
```

```
         objeto = tenta_alocar(geração_antiga, tamanho);
         se não_nulo(objeto)
            então retorna objeto;
         fim-se
      senão
         objeto = tenta_alocar(geração_jovem_origem, tamanho);
         se não_nulo(objeto)
            então retorna objeto;
         fim-se
      fim-se
      realiza_coleta();
   fim-enquanto
fim-função
```

Um trecho deste algoritmo foi alterado para poder suportar a alocação via lista de espaços livre. Inicialmente a lista encontra-se vazia. Após a realização da coleta, a lista é criada. Na próxima tentativa de alocação, um objeto não-nulo será retornado. A seguir, listamos o algoritmo alterado para o gerente de alocação:

```
função faz_alocação(tamanho) : endereço_memória
   gera_estatisticas();
   enquanto (verdade) faça
      objeto = tenta_alocar(geração_jovem_eden, tamanho);
      se não_nulo(objeto)
         então retorna objeto;
      fim-se
      adquiri_lock_heap();
      se não_vazia(lista) então
         objeto = requisição_memoria(lista, tamanho);
         se não_nulo(objeto)
            então retorna objeto;
         senão
            tipo_coletor = copia;
         fim-se
      fim-se
      se muito_grande(objeto) então
         objeto = tenta_alocar(geração_antiga, tamanho);
         se não_nulo(objeto)
            então retorna objeto;
         fim-se
      senão
         objeto = tenta_alocar(geração_jovem_origem, tamanho);
         se não_nulo(objeto)
            então retorna objeto;
         fim-se
      fim-se
      realiza_coleta();
   fim-enquanto
fim-função
```

A função requisição_memoria(lista, tamanho) verifica na lista qual é o melhor bloco livre que satisfaz a alocação do objeto com o tamanho especificado. Existem duas situações em que a lista não possui blocos que satisfaçam a requisição. Na primeira

situação, o objeto pode ser muito grande para ser alocado nesta geração. Neste caso tentamos alocá-lo na geração antiga. A segunda situação ocorre em função da inexistência de espaço contíguo disponível na geração jovem suficiente para satisfazer a requisição. Nessa situação é executado o algoritmo padrão que emprega cópia.

A função realiza_coleta() prepara a máquina virtual para a coleta, gerando sinais no gerenciador de eventos para que o aplicativo pare sua execução momentaneamente. No momento em que a JVM está preparada, o coletor da *DefNewGeneration* executa a função para coleta. Listamos, a seguir, o algoritmo da coleta alterado e em alto nível:

```
procedimento coletor()
   gera_estatisticas();
   limpar(lista);
   marca_heap();
   se (tipo_coletor = copia) então
      copia_objetos();
      tipo_coletor = mark_lazy_sweep;
   senão
      cria_lista();
   fim-se
   coleta_estatisticas();
fim-procedimento
```

No inicio do processo de coleta a lista atual é apagada, independente do coletor a ser executado. Caso o coletor de cópia seja executado a lista continua vazia, pois todo o Éden estará disponível para alocação após o coletor. Caso contrário, a lista é montada a partir dos blocos de espaços gerados pelos objetos não marcados. Como ilustrado na figura 16, a lista mantém o tamanho e o ponteiro para o início de cada bloco livre.

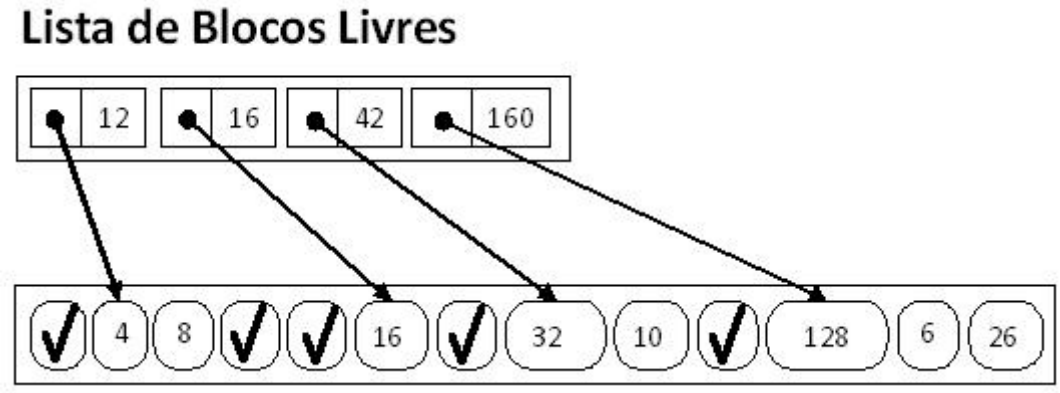

Figura 16 – Lista de Blocos Livres

Para construir a lista é feita uma varredura no Éden, verificando quais são os objetos não marcados. Objetos contíguos não marcados são considerados um único bloco de espaço cujo tamanho é a soma dos tamanhos destes objetos. O algoritmo em alto nível da função cria_lista() está descrito a seguir:

```
procedimento cria_lista()
   objeto = inicio_eden;
   enquanto (objeto < topo_eden) faça
     se não_marcado(objeto) então
        cria(bloco);
        enquanto não_marcado(objeto) E (objeto<topo_eden) faça
           adiciona_objeto(bloco, objeto);
           objeto = objeto + tamanho(objeto);
        fim-enquanto;
        adiciona_bloco(lista, bloco);
     fim-se
     objeto = objeto + tamanho(objeto);
   fim-enquanto
fim-procedimento
```

Ao final da criação da lista vários blocos de tamanhos variados estarão disponíveis para novas alocações.

Chegamos a iniciar a implementação deste algoritmo modificado de coleta de lixo na JVM. O algoritmo de criação da lista e alocação foi testado fora da JVM e funciona. Mas ao acoplá-los na JVM tivemos problemas que levavam a JVM a quebrar: algumas exceções do tipo *NullPointerException* são lançadas, apesar de todos os objetos criados pelo aplicativo terem sido inicializados corretamente. Isso nos faz suspeitar que a máquina não marca todos os objetos que estão vivos.

Analisando a implementação da JVM, verificamos que nem todos os objetos vivos são de fato marcados no processo de marcação dos objetos. Este fato vai de encontro a especificação da JVM, já que, segundo ela, todos os objetos deveriam ser marcados para realizar a coleta. Lembramos que utilizamos o processo de marcação original, sem modificações.

Como o algoritmo desenvolvido depende que todos os objetos vivos, sem exceção, estejam marcados, não conseguimos concluir a implementação deste algoritmo. Buscamos descobrir, sem sucesso, como a JVM resolve o problema destes objetos não marcados, já que a solução empregada pela JVM poderia ser também utilizada em nossa implementação.

4.3.2. Aumento Automático da *Heap*

Outra forma de reduzir o tempo de pausa do aplicativo seria reduzir a quantidade de coletas feitas durante sua execução. Para reduzir a quantidade de coleta deve-se reduzir a quantidade de falhas de alocação que podem ocorrer durante a execução do aplicativo. Por sua vez, para reduzir as falhas de alocação deve-se disponibilizar mais memória para a execução do aplicativo. Mas disponibilizar memória demais para o aplicativo pode ser um desperdício.

A JVM disponibiliza ao aplicativo um determinado tamanho de *heap* fixo durante toda a execução. Dependendo das características do aplicativo, aumentar o tamanho da *heap* pode se traduzir em uma redução do tempo de execução em função de uma menor quantidade de coletas.

Partindo-se deste fato, outra tentativa de implementação foi feita no Hotspot: um algoritmo foi desenvolvido com o intuito de ajustar o tamanho do Éden de acordo com uma heurística simples o suficiente para não demandar muito tempo de computação. Assim, o Éden é ajustado de acordo com as características do aplicativo em execução.

Como a JVM não suporta o aumento do tamanho da *heap* em tempo de execução, o tamanho inicial do Éden foi reduzido à metade do tamanho tradicionalmente empregado pela JVM. O Éden, neste caso, pode expandir-se no máximo até o tamanho padrão estabelecido pela JVM.

A heurística utilizada para o ajuste do Éden consiste em verificar o tempo entre certa quantidade de coletas. Se este tempo for relativamente pequeno, o tamanho do Éden é aumentado; se o tempo for relativamente grande, o tamanho do Éden é reduzido; caso contrário mantém-se o tamanho atual.

Se o tempo entre coletas for relativamente pequeno, então quer dizer que muitas alocações estão ocorrendo na aplicação. O aumento do Éden é realizado na tentativa de suprir essas alocações com menos coletas. Se o tempo entre coletas for relativamente grande, então poucas alocações estão ocorrendo, ou seja, não há a necessidade de ter tanta memória disponível para alocações. Reduzimos neste caso o tamanho do Éden, com o intuito de economizar a memória.

O processo de aumento e redução do Éden é feito na classe *EdenSpace* do *DefNewGeneration*. Essa classe contém o limite inferior e o superior do espaço delimitado para o Éden. Como ilustrado na figura 17, os campos *bottom*, *top* e *end* são usados para delimitar o Éden, sendo *bottom* e *end* fixos durante toda a execução da máquina virtual. O

campo *top* indica o ponto no qual um novo objeto pode ser alocado; todo o espaço à esquerda deste ponto contém objetos recém alocados. Quando valor do campo *top* alcança o valor do *end*, uma coleta é executada e todos os objetos vivos são retirados deste espaço, e valor do *top* é substituído pelo valor do *bottom*.

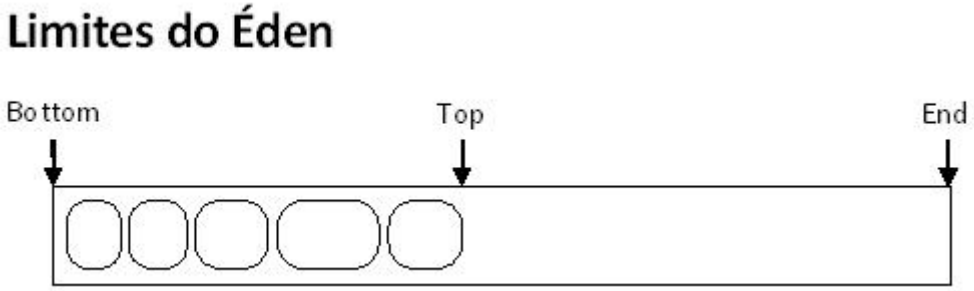

Figura 17 – Limites do Éden

Para que o ajuste de tamanho funcione, um novo campo chamado *hardEnd* foi adicionado na classe *EdenSpace* para armazenar o limite superior do Éden. Com isso, o valor do campo *end* pode variar entre *bottom* e *hardEnd*. Na inicialização do *EdenSpace*, os valores dos campos *end* e *hardEnd* são definidos, sendo o valor do *hardEnd* igual ao final do espaço e o valor do *end* o meio do espaço, como ilustra a figura 18.

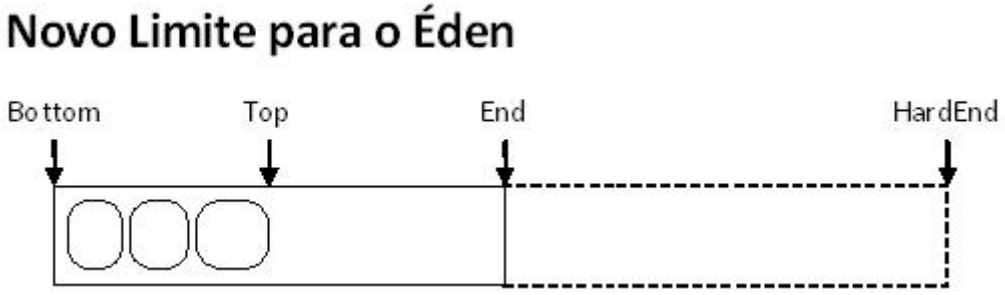

Figura 18 – Novo Limite para o Éden

A alteração do tamanho do Éden só é feita após a execução de uma coleta de lixo, pelo fato de que neste ponto o coletor já moveu todos os objetos vivos para outro espaço. Sendo assim, pode-se alterar o campo *end* para qualquer posição no intervalo definido pelo *bottom* e *hardEnd*.

A taxa de aumento do tamanho do Éden é sempre a metade do espaço disponível, ou seja:

$$end = end + \frac{(hardEnd - end)}{2} = \frac{hardEnd + end}{2}$$

Enquanto houver espaço disponível para realizar o aumento do Éden, o campo é alterado. Lembre-se que essa alteração só ocorre caso muitas coletas sejam executadas em um intervalo de tempo relativamente pequeno.

No caso de redução do tamanho do Éden, a taxa de redução é o valor do espaço disponível:

$$end = end - (hardEnd - end) = 2end - hardEnd$$

A fórmula acima é valida caso o espaço disponível não seja a metade do espaço total disponível para o Éden, pois assim o total de espaço para alocação se reduz a zero, causando falha na máquina virtual.

Outro problema é quando não há espaço disponível para o aumento do Éden, pois a taxa de reduz depende diretamente da quantidade de espaço disponível. Neste caso o tamanho do Éden é reduzido em ¼ do tamanho total.

Esses valores foram escolhidos através de teste feitos na JVM, em diversas aplicações. Esses testes estão descritos no próximo capitulo juntamente com os resultados obtidos.

Capitulo 5

Resultados Experimentais

Um benchmark (CHAVES, 2007) foi utilizado com o propósito de avaliar o desempenho da heurística implementada no coletor serial da geração jovem. O benchmark é composto pelos aplicativos GCOld, GCBench, JGFCreateBench, JGFSerialBench, JGFSORBench e o JGFMonteCarlo.

Os principais fatores avaliados em nossos testes foram o tempo de execução do aplicativo, o tempo total de pausa, o tempo médio de execução das coletas, taxa de *throughput* e o *footprint*.

O *throughput* é a quantidade de processamento gasto apenas para a execução do aplicativo, ou seja, neste contexto é a porcentagem de tempo que a JVM não executa coleta de lixo em relação ao tempo total de execução do aplicativo. O *footprint* é o uso máximo de memória do aplicativo.

Os testes foram realizados em uma máquina com processador AMD Sempron™ Processor 2800+ 32 bits com *clock* de 1.6 Ghz, com 128KB de *cache* L1 e 256KB de *cache* L2. A máquina em questão tem 1 Gb de memória principal e executa Linux Kubuntu 7.10 com *kernel* 2.6.22.4-386. Testamos duas versões do jre1.6.0 (*Java Runtime Machine*) compiladas na própria máquina, senda uma das versões sem alteração do código e a outra com a implementação da heurística.

Para avaliar os resultados obtidos geramos o log da coleta de lixo para cada uma das aplicações e o aplicativo *GCViewer* (Tagtraum industries, 2005) para interpretar estes arquivos.

5.1. Descrição dos Aplicativos

Cada aplicativo foi executado três vezes e a partir dos resultados uma média dos valores foi computada. Algumas variações quanto ao tamanho da *heap* foram feitas.

O GCOld tenta simular aplicações que mantém uma grande quantidade de objetos vivos durante um período longo de tempo, sempre criando novos objetos que permaneceram vivos tempo suficiente para serem promovidos para a geração antiga. Esta aplicação mantém um vetor de ponteiros para raízes de árvores binárias e sua execução é dividida em duas fases: inicialização e fase estável. Na inicialização é realizada a alocação das árvores. Na fase estável são executados vários passos: alocação de uma certa

quantidade de objetos que se tornam lixo no final de cada passo; alocação de uma árvore binária, que no final do passo substitue parte de alguma árvore alocada na inicialização; pequenas modificações nos dados contidos nos nós das árvores, chamadas de mutações. Esses passos são realizados várias vezes durante a execução da aplicação.

Os parâmetros de entrada são o tamanho (MB) da estrutura a ser mantido vivo (32 MB no nosso caso), o número de mutações (10 por passo), a proporção de objetos que ficarão armazenados na memória (2), o número de mudanças de ponteiros em cada passo (300) e por fim o número de passos (100 no total).

GCBench é uma aplicação que tenta simular o comportamento da maioria das aplicações reais no quesito de requisições de alocação de memória. Inicialmente vários objetos são alocados na tentativa de popular todo o espaço da geração jovem e antiga. Após ter preenchida toda a *heap*, o aplicativo passa a criar e destruir árvores binárias balanceadas de diferentes tamanhos. Esse aplicativo também mantém um vetor com elementos do tipo ponto flutuante, fazendo algumas operações com os elementos deste vetor.

Os parâmetros para este aplicativo são a profundidade da árvore que será mantida ativa na memória, o tamanho do vetor e os limites (superior e inferior) para as profundidades das várias árvores que são construídas e destruídas. Nos testes realizados foram utilizadas as mesmas entradas utilizadas em (CHAVES, 2007): uma árvore de 20 níveis de profundidade; uma árvore de profundidade 18 e um vetor com 5.000.000 posições que serão mantidos em memória e testes com árvores de profundidade entre 4 e 24 níveis, considerando apenas os valores pares.

Os outros aplicativos executados fazem parte do Java Grande *Suite Benchmark* (EPCC, 2007). Do conjunto de benchmarks disponíveis pelo Java Grande, apenas a versão serial de cada aplicativo foi executado.

O JGFCreateBench faz a alocação de objetos de tipos distintos. O aplicativo simplesmente executa um loop, fazendo várias alocações em uma mesma variável. Iisso implica que esses objeto alocados irão ter um tempo de vida muito curto, sendo reciclados nas coletas executadas. O total de objetos alocados chega a casa de 10^8.

O JGFSerialBench realiza várias operações de entrada e saída, criando e carregando dados em arquivos. Para efetuar essas tarefas ele utiliza estruturas da API do Java para armazer e carregar esses dados. As estruturas usadas representam uma coleção de dados

que são gravados e carregados em arquivo, para cada uma dessa estruturas são realizadas aproximadamente 4000 iterações gravando e lendo os arquivos.

O JGFSORBench faz 100 iterações do algoritmo SOR (*Successive over relaxation*, método numérico utilizado para aumentar a convergência do método de *Gauss-Seidel* para resolução de sistemas de equações lineares) em uma matriz de dados. O aplicativo realiza algumas interações sobre os elementos de uma matriz com o objetivo de atualizar um elemento principal e os elementos vizinhos do mesmo, incluindo os elementos previamente atualizados. A entrada deste aplicativo é uma matriz de 2000x2000 elementos representando um sistema linear de equações.

O JGFMonteCarlo realiza uma simulação financeira, utilizando a técnica de Monte Carlo para definição de preços de produtos com base em fatores distintos. O código gera algumas séries temporais com as mesmas variações dos dados de uma série histórica de informações. A entrada para a execução dos testes é um vetor de dados inteiros com 6000 posições.

Os teste foram realizados tanto no compilador *client* quanto no s*erver*, o tamanho total do *heap* foi de 256 MB, variando apenas o tamanho da geração jovem de três formas distintas: 32 MB fixo, com tamanho inicial 32 MB e máximo de 64 MB e por fim com 64 MB fixo. Com essas opções, os aplicativos descritos anteriormente foram executados na JVM original. Para a JVM alterada com a heurística descrita no capitulo anterior a geração jovem foi iniciada com 32 MB. A heurística utilizada foi a de aumentar o tamanho do Éden após 30 coletas menores seguidas executadas em intervalo de tempo de 100 milissegundos.

5.2. Resultados

5.2.1. GCOld

A característica marcante deste aplicativo é a criação de muitos objetos grandes que são mantidos alocados durante muito tempo. Assim a geração antiga é muito utilizada, o que faz com que coletas completas sejam realizadas com mais freqüência. Como nossa heurística tenta reduzir a quantidade de coletas menores, esperamos não ter ganho algum neste benchmark. De fato, observando os resultados da tabela 1 percebemos que o uso da heurística na geração jovem não foi de grande valia.

Apesar de não ter nenhum ganho quanto ao tempo de execução, por outro lado não adicionamos mais custos a este.

Tabela 1 - Resultados do aplicativo GCOld no compilador *client*

Execução realizada com heap total de 256 mb

				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Client - 32 mb	29,35	103,12	11,450	0,18467	16,00
2	Client - 32 -> 64 mb	28,77	102,88	11,440	0,18457	16,00
3	Client - 64 mb	31,85	141,5	10,420	0,33611	15,00
4	Client - Com heuristica	26,82	119,781	11,920	0,23367	16,00

Os resultados obtidos nos testes para o compilador *server* foram semelhantes aos obtidos para o compilador *client*.

5.2.2. GCBench

Essa aplicação tem como característica principal criar uma grande quantidade de objetos com tempo de vida curto, fazendo uma grande quantidade de alocações em toda a sua execução, o que provoca várias coletas menores.

Tabela 2 - Resultados do aplicativo GCBench no compilador *client*

Execução realizada com *heap* total de 256 mb

				Tempos (seg)		
	Compilador - tamanho	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Client - 32 mb	36,19	145,55	3,687	0,05263	5,00
2	Client - 32 -> 64 mb	36,17	145,55	3,690	0,05276	5,00
3	Client - 64 mb	73,94	130,78	0,750	0,02151	2,00
4	Client - Com heuristica	43,31	163,344	2,570	0,04585	4,00

Observando os dados coletados da execução do aplicativo na tabela 2, pode-se perceber que quanto maior a quantidade de memória disponibilizada para o aplicativo, menor é o tempo acumulado (correspondente ao tempo total gasto com coletas), resultando em uma redução no tempo total de execução do aplicativo.

A configuração 1 representa a pior execução, considerando o tempo como variável de comparação; do tempo total da execução do aplicativo aproximadamente 36% foram gasto com a execução do aplicativo, ou seja, o custo da execução dos coletores foi muito maior do que o da execução do aplicativo.

Como pode ser observado na configuração 3, executando o aplicativo com a 64 MB para a geração jovem, o tempo de execução foi o melhor em comparação a todos os outros testes. Isso aconteceu pelo fato da quantidade de coletas realizadas serem bem menor, cerca de 26% do tempo, como mostra o gráfico da figura 13.

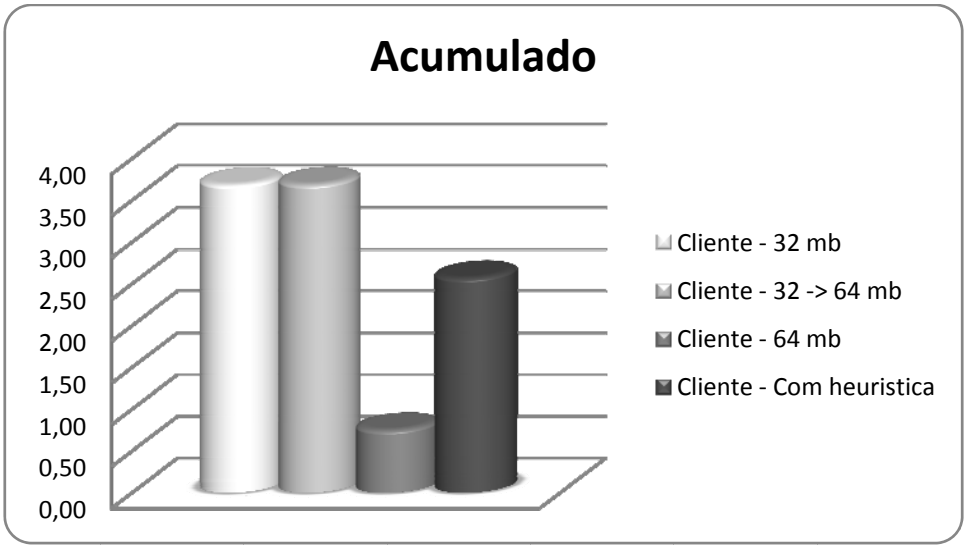

Figura 13 – GCBench, tempo total gasto com coletas. (Acumulado)

Com relação à heurística implementada na JVM, tem-se uma redução no tempo total de execução do aplicativo em comparação as execuções com 32 MB de tamanho fixo e o de 32 MB inicial com máximo de 64 MB, mas não tão bom quanto ao de 64 MB fixo. Neste caso comprova-se que com o aumento do tamanho do Éden, o número de coletas reduz, diminuindo o tempo de execução do aplicativo. O algoritmo com heurística reduziu o tempo de execução do aplicativo em 20%, mas com um custo na utilização de memória, pois na tentativa de adaptar o tamanho do Éden foi necessário utilizar 12% a mais de memória do que os testes das configurações 1 e 2. Esse custo é causado pelo fato de manter mais lixo na memória, pois como a idéia é reduzir o número de coletas, com certeza para esta aplicação a quantidade de lixo será maior.

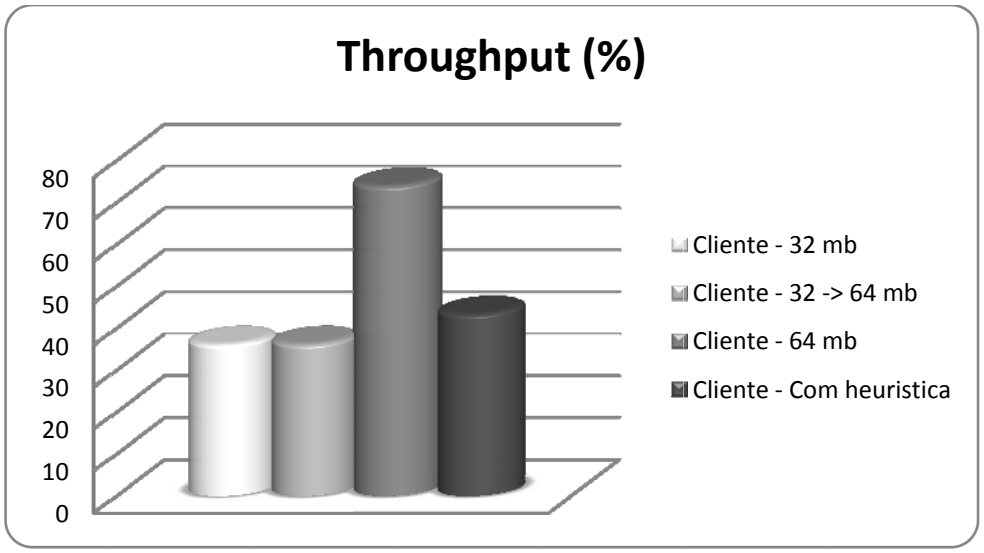

Figura 14 – GCBench, taxa de Throughput (%)

Observando a taxa de *throughput* pode-se concluir, com o auxilio do gráfico da figura 14, que em aplicativos em que se necessita de muita memória, o ideal é aumentar o tamanho da geração jovem. Entretanto, em sistemas com pouca memória, como sistemas embarcados, isso pode ser uma tarefa complicada.

Os resultados obtidos para o compilado *server* foram bem próximos dos obtidos pelo compilador *client*, como pode ser observado na tabela 3. Os tempos de execução do aplicativo foram similares, assim como o *footprint* e *throughput*, ou seja, as conclusões tiradas dos testes para o compilador *client* são válidas para o compilador s*erver*.

Tabela 3 - Resultados do aplicativo GCBench no compilador *server*

Execução realizada com heap total de 256 mb				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Server - 32 mb	38,03	145,52	3,7033	0,05	5,00
2	Server - 32 -> 64 mb	38,09	145,52	3,7033	0,05292	5,00
3	Server - 64 mb	72,20	130,78	0,7600	0,02161	2,00
4	Server - Com heuristica	45,55	163,336	2,5300	0,04525	4,00

5.2.3. JGFCreateBench

Esse aplicativo tem a mesma característica do GCBench, com a diferença de que a quantidade de objetos que se tornam lixo mais cedo é maior. Nesse aplicativo espera-se que uma grande quantidade de coletas menores seja feita. Assim, a expectativa é que os resultados sejam semelhantes ao GCBench.

Os resultados obtidos estão descritos na tabela 4. Pode-se perceber que os resultados para execuções com o tamanho da geração jovem fixo foram piores do que com o tamanho variável. Neste caso, o uso da heurística reduziu o tempo total em 7%, mas o tempo acumulado gasto com coletas foi o maior, apesar da diferença ser muito pequena. A média de pausa foi a pior também, com uma diferença muito pequena em comparação com as execuções com tamanho fixo. Conclui-se então que a heurística conseguiu reduzir a quantidade de coleta, que era seu objetivo inicial.

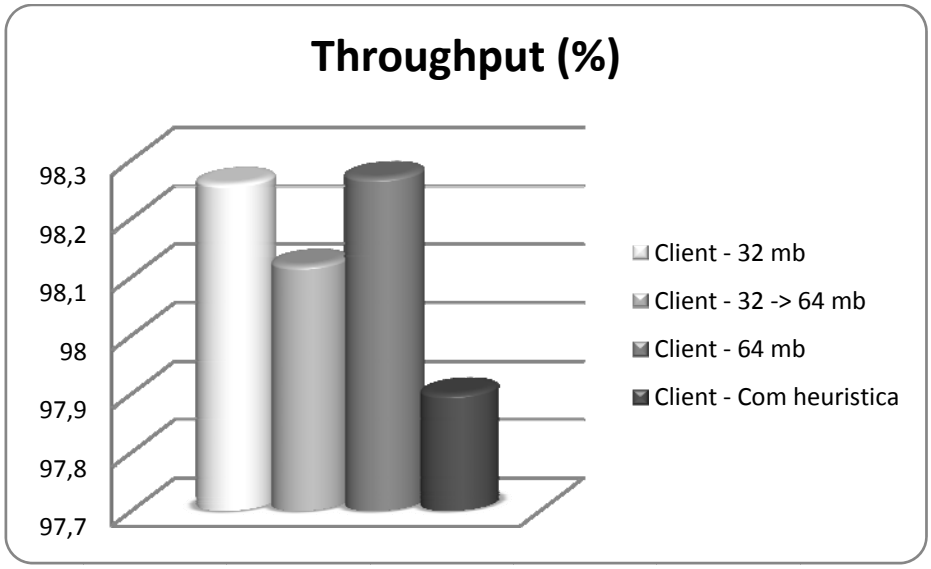

Figura 15 – JGFCreateBench, taxa de Throughput (%)

Uma coisa interessante que pode ser observando nos resultados é que mesmo tendo uma taxa de *throughput* (figura 15) um pouco mais baixa e um tempo acumulado ligeiramente maior, a heurística conseguiu reduzir o tempo total de execução do aplicativo. Isso pode ter ocorrido pelo fato da heurística disponibilizar ao aplicativo espaço em momentos críticos de alocação, o que não ocorre nas execuções com o tamanho fixo. Pode-se perceber que a execução com tamanho inicial 32 e máximo 64 obteve o mesmo resultado no que diz respeito a redução do tempo da execução, devido ao mesmo motivo da heurística.

Os resultados obtidos na execução deste teste no compilador *server* seguem o mesmo padrão do *client*, com a diferença de que o tempo total de execução do aplicativo reduziu consideravelmente. Isso ocorreu pelo fato do compilador *server* otimizar os *bytecodes* com o intuito de reduzir tempo final de execução.

Tabela 4 – Resultados do aplicativo JGFCreateBench no compilador *client*

Execução realizada com heap total de 256 mb				**Tempos (seg)**		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Client - 32 mb	98,26	32,812	0,9600	0,00056	55,00
2	Client - 32 -> 64 mb	98,12	32,812	0,9600	0,00056	51,00
3	Client - 64 mb	98,27	61,625	0,9600	0,00111	55,00
4	Client - Com heuristica	97,9	61,6	1,0700	0,01170	51,00

Tabela 5 - Resultados do aplicativo JGFCreateBench no compilador *server*

Execução realizada com heap total de 256 mb

	Coletor e opções	*Throughput (%)*	*Footprint (MB)*	Tempos (seg) Acumulado	Média pausa	Execução
1	Server - 32 mb	97,97	32,812	0,9500	0,00049	46,00
2	Server - 32 -> 64 mb	97,87	32,812	0,9800	0,00051	45,00
3	Server - 64 mb	98,17	61,625	0,8400	0,00085	46,00
4	Server - Com heuristica	97,90	61,612	0,9500	0,00093	45,00

Outra característica que pode ser observada nestes resultados é que apesar das configurações 4 (heurística) e 2 (*heap* variando de 32 a 64 MB) terem obtido os mesmos tempos de execução, a configuração 2 consumiu menos memória que a configuração 4. Isso ocorreu porque o objetivo da heurística era reduzir a quantidade de coletas. Por esse motivo uma quantidade maior de lixo é mantida na memória por mais tempo, fato que não ocorreu na configuração 2.

5.2.4. JGFSerialBench

Este aplicativo trabalha com grandes quantidades de dados que são carregados e armazenados em arquivos. A expectativa para este aplicativo é que utilize muita memória e com isso realize muitas coletas, o que pode ser interessante para nossa heurística.

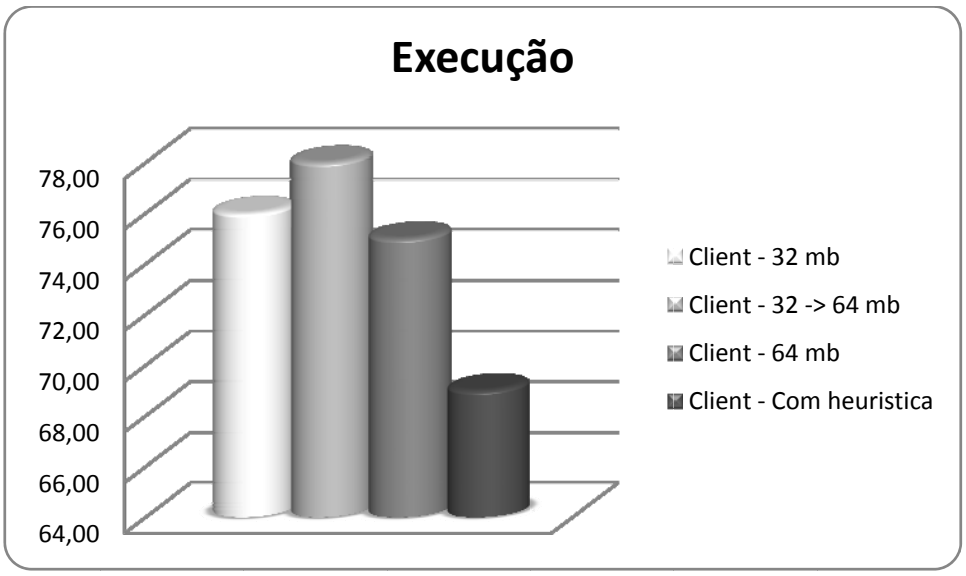

Figura 16 – JGFSerialBench, tempo total de execução

Para este aplicativo obteve-se ótimos resultados para a heurística na questão da redução do tempo total de execução (figura 16), mais de 11% em comparação ao pior resultado. Isso aconteceu por causa das características desse aplicativo, que aloca uma grande quantidade

de memória. A heurística disponibiliza memória para o aplicativo na fase de alocação dos objetos, reduzindo a quantidade de coletas. Interessante é que o uso total de memória (*footprint*) foi menor na heurística que na execução dos outros testes, como ilustrado na figura 17.

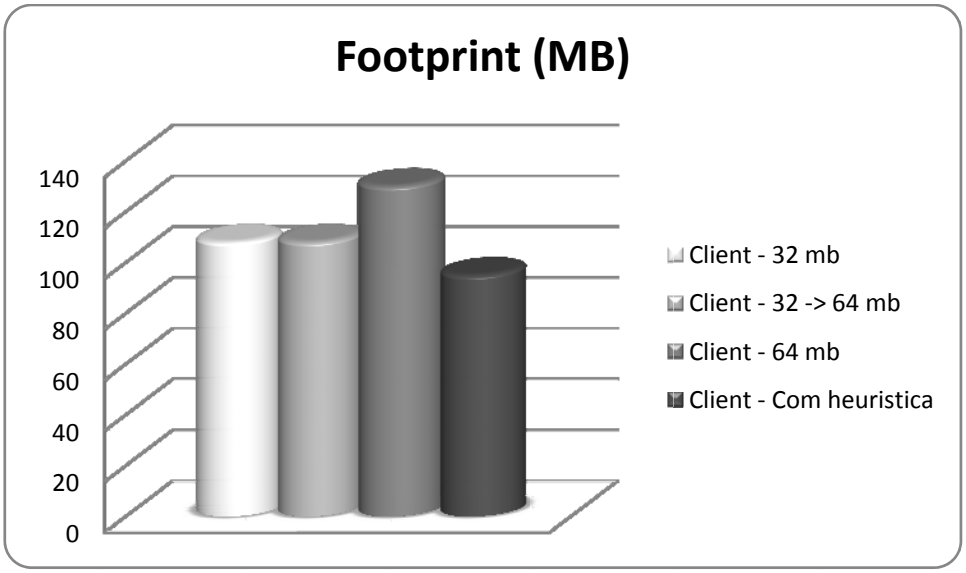

Figura 17 – JGFSerialBench, *footprint* (MB)

Tabela 6 – Resultados do aplicativo JGFSerialBench no compilador *client*

Execução realizada com heap total de 256 mb

				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Client - 32 mb	95,92	108,084	3,1200	0,14851	76,00
2	Client - 32 -> 64 mb	96,05	108,086	3,1000	0,14784	78,00
3	Client - 64 mb	97,15	130,055	2,1700	0,21711	75,00
4	Client - Com heuristica	96,09	94,969	2,7200	0,12930	69,00

Os resultados obtidos nos testes no compilador *server* foram semelhantes aos obtidos no compilado *client*. A única diferença marcante é que com o uso da heurística o total de memória usada pelo aplicativo foi muito maior no compilador *server* que no *client*.

Tabela 7 – Resultados do aplicativo JGFSerialBench no compilador *server*

Execução realizada com heap total de 256 mb

				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Server - 32 mb	95,70	156,758	3,2000	0,15208	75,00
2	Server - 32 -> 64 mb	96,11	156,758	3,1700	0,15109	81,00
3	Server - 64 mb	97,34	154,484	1,9100	0,19124	72,00
4	Server - Com heuristica	95,31	164,445	3,1800	0,15166	68,00

5.2.5. JGFSORBench

Esse aplicativo realiza muito processamento, alocado os dados necessários no inicio da execução. Espera-se que poucas coletas sejam realizadas e, assim como GCOld, nossa heurística não apresente vantagem.

A configuração com tamanho da geração jovem de 64 fixo obteve os melhores resultados, mas a diferença em comparação as demais configurações é mínima. Os resultados são praticamente iguais também para as duas versões da máquina Java, *client* e *server*.

Com era de se esperar, em aplicativos que realizam poucas operações de coleta de lixo, o uso da heurística não é pior nem melhor que as demais configurações, mostrando que a heurística não possui custos adicionais de implementação.

Novamente notamos que o footprint foi menor com a nossa heurística, quando comparada as versões que utilizam uma mesma quantidade de memória (32 MB).

Tabela 8 – Resultados do aplicativo JGFSORBench no compilador *client*

Execução realizada com heap total de 256 mb				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Client - 32 mb	98,97	65,684	0,0800	0,03769	7,00
2	Client - 32 -> 64 mb	98,97	65,684	0,0800	0,03774	7,00
3	Client - 64 mb	99,86	61,625	0,0100	0,01003	7,00
4	Client - Com heuristica	98,93	63,656	0,0800	0,03924	7,00

Tabela 9 – Resultados do aplicativo JGFSORBench no compilador *server*

Execução realizada com heap total de 256 mb				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Server - 32 mb	98,97	65,684	0,0700	0,03649	7,00
2	Server - 32 -> 64 mb	98,95	65,684	0,0700	0,03686	7,00
3	Server - 64 mb	99,86	61,625	0,0100	0,00951	7,00
4	Server - Com heuristica	98,92	63,656	0,0800	0,03821	7,00

5.2.6. JGFMonteCarlo

Esse algoritmo tem as mesmas características do JGFSORBench. Por manter um mesmo conjunto de dados vivos durante toda a execução, espera-se que poucas coletas sejam realizadas. Assim, esperamos resultados semelhantes para este benchmark: tempos semelhantes aos das demais aplicações. De fato, os resultados das tabelas 10 e 11 confirmam isso.

Tabela 10 – Resultados do aplicativo JGFMonteCarlo no compilador *client*

Execução realizada com heap total de 256 mb

				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Client - 32 mb	92,97	104,406	0,4200	0,03521	5,00
2	Client - 32 -> 64 mb	92,83	104,406	0,4100	0,03455	5,00
3	Client - 64 mb	93,22	137,016	0,4200	0,06927	6,00
4	Client - Com heuristica	92,05	108,32	0,5000	0,03873	6,00

Tabela 11 – Resultados do aplicativo JGFMonteCarlo no compilador *server*

Execução realizada com heap total de 256 mb

				Tempos (seg)		
	Coletor e opções	***Throughput (%)***	***Footprint (MB)***	**Acumulado**	**Média pausa**	**Execução**
1	Server - 32 mb	93,41	104,211	0,4200	0,03503	6,00
2	Server - 32 -> 64 mb	93,48	104,211	0,4100	0,03455	6,00
3	Server - 64 mb	93,54	137,051	0,4200	0,06924	6,00
4	Server - Com heuristica	92,36	108,285	0,5000	0,03836	6,00

Capitulo 6

Conclusão

Conforme mostramos ao longo do Capítulo 5, nossa heurística mostrou-se uma boa alternativa para aplicativos que utilizam uma grande quantidade de memória por pouco tempo. Mesmo para aplicativos que utilizam objetos por um longo período de tempo, sem que sejam realizadas coletas de lixo, tem vantagem no uso de nossa heurística: sem que sejam adicionados custos em termos de tempo de computação, o *footprint* foi reduzido.

Se uma heurística desse tipo fosse implementada na JVM utilizando mais dados estatísticos colhidos pela própria máquina, talvez seu funcionamento fosse bem melhor do que com a heurística atual.

Acreditamos que o objetivo deste trabalho foi alcançado, uma vez que implementamos com sucesso uma heurística para a coleta de lixo na geração jovem da JVM. Lamentamos apenas não ter havido tempo para concluirmos nossa implementação original, que visava implementar uma versão modificado do coletor *MarkLazySweep*.

Trabalhos futuros envolvendo coleta de lixo na JVM poderiam:

- Tentar criar uma heurística para a geração antiga, na tentativa de ajustar seu tamanho de acordo com as características do aplicativo, da mesma forma feita para a geração jovem. Isso poderia ter bons resultados em aplicativos que mantém dados alocados por um período longo de tempo, como é o caso do GCOld;
- Retirar o limite fixo de aumento de memória da heurística, tentando adquirir do sistema operacional memória de acordo com a demanda, tanto para a geração antiga quanto para a jovem. Mas para isso devem ser feitas alterações muito mais complexas no código da JVM, de forma a permitir o aumento ilimitado da *heap*.
- Concluir a implementação do coletor *MarkLazySweep*.

Referências Bibliográficas

BACON, David, Perry CHENG, e V RAJAN. “A Unified Theory of Garbage Collection.” *Conferencia sobre programação orientada à objetos, sistemas, linguagens e aplicações.* Nova Iorque: ACM Press, 2004. 50-68.

BOEHM, Hans-J, Alan J. DEMERS, e Scott SHENKER. “Mostly parallel garbage collection.” *SIGPLAN (Simpósio de Linguagem de Programação Design e Implementação).* Toronto, 1991. 157-164.

CHAVES, Luciano Jerez. *Análise de Desempenho das Implementações de Coleta de Lixo da Máquina Virtual Java.* Juiz de Fora: Universidade Ferderal de Juiz de Fora, 2007.

DETLEFS, D. http://192.18.108.226/ECom/EComTicketServlet/BEGIN2AA579EA8800A237BE9DF8AAB2DDA84E/-2147483648/2469782379/1/547718/547706/2469782379/2ts+/westCoastFSEND/ES-Gcold-1.0-G-F/ES-Gcold-1.0-G-F:1/GCold-1.0.tar.gz (acesso em 1 de dezembro de 2007).

EPCC. *The Java Grande Forum Benchmark Suite.* 2007. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html (acesso em 1 de dezembro de 2007).

GOENZ, Brian. *Java theory in pratice: a brief history of garbage collection.* 28 de outubro de 2003. http://www.ibm.com/developerworks/java/library/j-jtp10283/ (acesso em 10 de agosto de 2007).

HARRIS, Timothy. “Early storage reclamation in a tracing garbage collector.” *SIGPLAN.* Nova Iorque: ACM Press, 1999. 46-53.

HENRIKSSON, Roger. *Adaptive scheduling of incremental copying garbage collection for interactive applications.* Relatório Tecnico, Suécia: Lund University, 1996.

HERTZ, Matthew, Yi FENG, e Emery BERGER. “Garbage Collection without paging.” *Conferencia sobre projeto de linguagem de programação e implementação.* Nova Iorque: ACM Press, 2005. 143-153.

HUDSON, Richard L., e J. Eliot B. MOSS. “Incremental Collection of Mature Objects.” *Workshop internacional sobre gerenciamento de memória*, setembro de 1992: 388-403.

JOHNSTONE, Mark Stuart. *Non-Compacting Memory Allocation and Real-Time Garbage Collection.* Dissertação Doutorado, Austin: University of Texas, 1997.

Levine, John, e Margy Levine Young. *IECC. GC FAQ - Algorítmos.* 24 de junho de 2007. http://www.iecc.com/gclist/GC-algorithms.html (acesso em 10 de agosto de 2007).

PRINTEZIS, Tony. *Garbage Collection in Java Programming Language: Overview, Techniques, Seccesses.* Palestra Sun Microsystems, Sun Microsystems, 2005. *Garbage Collection in the Java HotSpot Virtual Machine.* setembro de 2004. http://www.devx.com/java/Article/21977 (acesso em 20 de agosto de 2007).

RAVENBROOK. *The Memory Management Reference.* 2001. http://www.memorymanagement.org/ (acesso em 10 de agosto de 2007).

SOBALVARRO, Patrick G. *A lifetime-based garbage collector for LISP systems on general-purpose computers.* Relatório Técnico, Cambridge: Massachusetts Institute of Technology, 1988.

Sun Microsystems, Inc. *Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine.* 20 de fevereiro de 2003. http://java.sun.com/docs/hotspot/gc1.4.2/example.html (acesso em 2 de dezembro de 2007).

Tagtraum industries, inc. *tagtraum industries.* 2005. http://www.tagtraum.com (acesso em 28 de novembro de 2007).

VENNERS, Bill. *Inside the Java Virtual Machine.* http://www.artima.com/insidejvm/ed2/index.html (acesso em 10 de agosto de 2007).

WILSON, Paul. “Uniprocessor garbage collection techniques.” *Workshop sobre gerenciamento de memória.* Saint-Malo: Springer-Verlag, 1992.

WILSON, Paul, e Mark S JOHNSTONE. “Real-Time Non-Copying Garbage Collection.” *Conferencia sobre programação orientada à objetos, sistemas, linguagens e aplicações.* Washington: ACM Press, 1993.

WILSON, Paul, Michael LAM, e Thomas MOHER. “Caching considerations for general garbage collections.” *Conferencia sobre LISP e programação funcional.* São Francisco: ACM Press, 1992. 32-42.

WITHINGTON, P. T. “How Real is "Real Time"' Garbage Collection?” *Conferencia sobre programação orientada à objetos, sistemas, linguagens e aplicações.* Phoenix: AMC SIGPLAN, 1991.

Yang, T., Berger, E. D., Kaplan, S. F., Moss, J. E., & Moss, B. “CRAMM: virtual memory support for garbage-collected applications.” *Simpósio sobre projeto de sistemas operacionais e implementações.* Seattle: USENIX, 2006. 8-8.