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

Héberte Fernandes de Moraes[1], Luciano J. Chaves[2], Marcelo Lobosco[3]

[1]Programa de Engenharia de Sistemas e Computação – COPPE/UFRJ
Caixa Postal 68.511 – 21941-972 – Rio de Janeiro – RJ – Brasil

[2]Instituto de Computação – Universidade Estadual de Campinas
Caixa Postal 6176 – 13.084-971 – Campinas – SP – Brasil

[3]Departamento de Ciência da Computação – Universidade Federal de Juiz de Fora
Campus Universitário – 36.036-330 – Juiz de Fora – MG – Brasil

brebete@cos.ufrj.br, luciano.chaves@students.ic.unicamp.br,
marcelo.lobosco@ufjf.edu.br

Abstract. *The Garbage Collector mechanism frees the programmer from manually dealing with memory management. This is a standard feature of several programming languages, such as Java, but imposes an additional cost on program execution. An attempt to minimize this cost is to increase the size available for allocations, but memory can be wasted. If too little memory is available for allocation, the number of garbage collections increases, which in turn impacts negatively the computation time. In this work we present, implement and evaluate a new heuristic that aims to adjust automatically the total amount of memory available for allocation. The heuristic was implemented in the Java Virtual Machine.*

Resumo. *O mecanismo de coleta automática de lixo (garbage collection), presente em diversas linguagens de programação, como Java, por um lado libera o programador da tarefa de gerenciar as regiões de memória que sua aplicação não mais utiliza, e por outro adiciona um custo ao tempo total de execução da aplicação. Uma tentativa de minimizar este custo consiste em aumentar o tamanho do espaço disponível para o aplicativo realizar alocações, o que por sua vez pode acarretar em um desperdício de memória. Sob outra perspectiva, a restrição demasiada do espaço disponível para alocação pode causar muitas coletas, conseqüentemente aumentando o tempo total de execução da aplicação. Neste trabalho propomos, implementamos e avaliamos uma nova heurística para adequar automaticamente a quantidade de memória disponível para alocação pelas aplicações que executam na Máquina Virtual Java.*

1. Introdução

O coletor de lixo (*Garbage Collector*) foi desenvolvido com o intuito de auxiliar o programador no desenvolvimento de suas aplicações. Em um primeiro momento, cabe ao coletor a tarefa de vasculhar a memória, procurando por regiões alocadas que não podem mais ser acessadas pela aplicação. Estas regiões são denominadas de lixo. Após

essa fase inicial, o coletor deve liberar essas regiões de memória, permitindo assim o reuso do espaço para novas alocações. Desta forma, o coletor gerencia automaticamente a utilização da memória por parte do aplicativo. Em linguagens de programação que não contam com este recurso, este papel é desempenhado pelo programador, que é obrigado a especificar explicitamente quais regiões devem ser liberadas. É comum que os programadores incorram em erros durante esse processo, que podem levar a erros fatais na execução dos programas. Com o coletor, o programador fica livre dessa responsabilidade, levando a um gerenciamento mais eficiente dos recursos de memória.

O emprego de coletores em linguagens de programação implica em um custo adicional para a aplicação, já que parte do seu tempo de execução é gasto nos processos de identificação e liberação de regiões de memória não mais alcançáveis pela aplicação. Esse custo pode ser muito alto, dependendo das características do aplicativo e do coletor utilizado. Uma forma de minorar ou mesmo eliminar este custo consiste do aumento da quantidade de espaço para o aplicativo realizar alocações: com mais espaço disponível, reduz-se a necessidade de liberar memória para o reuso da própria aplicação. Entretanto, disponibilizar muita memória pode constituir-se em fonte de desperdício de recursos computacionais. De forma análoga, restringir demais a quantidade de espaço disponível pode levar a ocorrência de muitas coletas de lixo, conseqüentemente aumentando o tempo total de execução da aplicação. Faz-se, portanto, necessário atingir um ponto de equilíbrio na quantidade de memória disponibilizada para aplicação.

Neste trabalho estudamos esta questão, propondo uma heurística para realizar automaticamente o aumento ou a redução da quantidade de memória disponibilizada para a aplicação. Para avaliar nossa proposta, implementamos a heurística na Máquina Virtual Java (*Java Virtual Machine* – JVM) [Lindholm e Yellin 1999], comparando-a com os coletores disponibilizados pela JVM. Os resultados preliminares demonstram que, para algumas aplicações avaliadas, a heurística foi efetiva em seu objetivo de reduzir os custos associados à coleta de lixo.

O restante deste trabalho está organizado da seguinte forma: a seção 2 apresenta os mecanismos de coleta de lixo disponibilizados na JVM; a seção 3 apresenta nossa heurística; na seção 4 são apresentados alguns aspectos de sua implementação na JVM; a seção 5 apresenta alguns resultados preliminares da heurística ora proposta; a seção 6 apresenta algumas idéias para trabalhos futuros; enquanto a seção 7 conclui o trabalho.

2. Coleta de Lixo na JVM

Existem diversos tipos de algoritmos para coleta de lixo. Segundo Bacon *et. al.* (2004), os algoritmos existentes podem ser divididos basicamente em dois grandes grupos: a) algoritmos de contagem de referências e b) algoritmos de rastreamento. Os algoritmos de contagem de referências armazenam, para cada objeto, um contador de referência. Esse contador é incrementado sempre que uma nova referência a este objeto é feita. Esse contador é decrementado caso uma dessas referências deixe de existir. Quando o valor do contador chega a zero, o objeto é considerado lixo e é reciclado. Já os algoritmos de rastreamento percorrem todas as referências de memória a partir de um conjunto inicial de referências, denominado conjunto raiz. Todos os objetos direta ou indiretamente alcançáveis a partir do conjunto raiz são mantidos em memória, enquanto os demais são reciclados.

Para a gerência de memória a JVM utiliza um mecanismo de rastreamento baseado em gerações. Esse mecanismo consiste do agrupamento dos objetos vivos em três distintas regiões de memória: a) geração jovem, b) geração antiga ou estável e c) a geração permanente. Esta divisão parte do pressuposto de que a maioria dos objetos vive por um período muito curto, enquanto uma pequena porcentagem deles vive por muito tempo [Ungar 1984]. Assim pode-se realizar operações de coleta mais constantes apenas na região de memória que abriga os objetos alocados mais recentemente, que ficam localizados na geração jovem. Após um determinado número de coletas, os objetos sobreviventes são promovidos para a geração antiga, conforme ilustrado na Figura 1. Nesta região de memória as coletas não precisam ser tão freqüentes quanto na geração anterior, visto que a chance dos objetos serem referenciados por muito tempo é maior. Na geração permanente são alocados meta-dados, como descritores de classes e métodos. As coletas na geração jovem são conhecidas por coletas menores. Quando as gerações antigas e permanente ficam sem espaço, ocorrem as coletas maiores. Uma vantagem deste modelo de coleta baseado em gerações é a possibilidade de utilizar técnicas de coleta de lixo distintas para cada uma das gerações, de acordo com as suas particularidades.

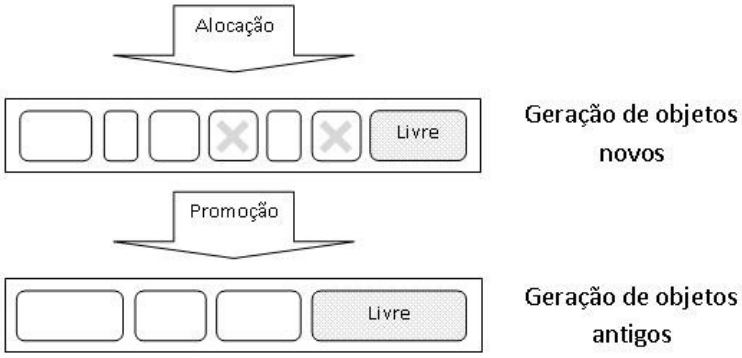

Figura 1. Fluxo de movimentação dos objetos

A geração jovem na JVM é ainda dividida em três sub-regiões: Éden, espaço de sobreviventes de origem e espaço de sobreviventes de destino, conforme ilustra a figura 2. Todas as novas alocações são feitas no Éden. Objetos que sobrevivem a primeira coleta no Éden são copiados para o espaço de sobreviventes de origem. Quando coletas precisam ser realizadas neste espaço, os objetos ainda vivos são copiados para o espaço de sobreviventes de destino. No final da cópia, os espaços de sobreviventes trocam de nomes.

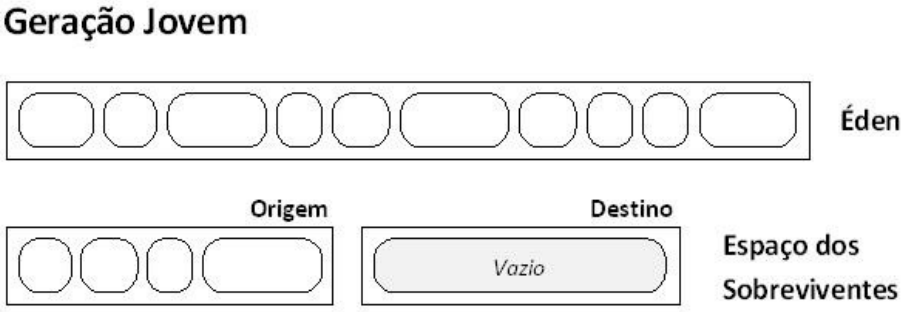

Figura 2. Modelo da geração jovem da JVM

3. Heurística Proposta

O coletor de lixo é considerado um custo adicional na execução dos aplicativos. Para reduzir este custo deve-se reduzir o tempo e / ou o número de execuções do coletor, já que quanto menor for o tempo gasto com coletas, menor será o tempo total de execução do aplicativo.

Para reduzir o tempo de execução de um coletor devemos levantar seus custos de execução. Baseados nessa informação, procuram-se alternativas de implementação para reduzir os custos das funções que demandam maiores tempos de execução. Por exemplo, no caso do coletor da geração jovem, o maior custo está relacionado às operações de cópia dos objetos vivos.

Outra forma de reduzir o tempo total de execução do aplicativo é diminuir a quantidade de coletas. Uma das formas de se fazer isto é aumentando a memória disponível para alocação. Entretanto, estaremos desperdiçando memória caso uma quantidade excessiva desta seja disponibilizada para a aplicação. Ficamos, assim, com um dilema: caso uma quantidade insuficiente de memória seja disponibilizada, teremos uma grande quantidade de coletas, aumentando o tempo total de execução dos aplicativos. Disponibilizar muita memória, por outro lado, pode constituir-se em desperdício de recursos computacionais.

Neste trabalho propomos uma heurística com o intuito de ajustar automaticamente o tamanho do Éden de acordo com as características de consumo de memória do aplicativo em execução. A heurística 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. A heurística se baseia no pressuposto de que se o tempo entre as coletas for relativamente pequeno, então muitas alocações estão ocorrendo na aplicação. O aumento do Éden é então realizado na tentativa de suprir essas alocações com mais memória, sem que sejam necessárias operações de coleta para disponibilizá-la. Por outro lado, se o tempo entre as coletas for relativamente grande, então poucas alocações estão ocorrendo, ou seja, não há a necessidade de haver tanta memória disponível para alocações. Neste caso reduzimos o tamanho do Éden com o intuito de economizar memória.

Nossa heurística inspeciona particularmente o Éden por esta ser a região onde a maior parte das coletas ocorre, as coletas menores, mas nada impede que a técnica seja empregada nas demais regiões de memória para diminuir o número de coletas maiores.

4. Aspectos de Implementação da Heurística na JVM

A implementação da heurística proposta foi realizada no *Hotspot* JVM, uma implementação da especificação da máquina virtual Java desenvolvida pela Sun Microsystem.

Uma das dificuldades que encontramos para implementar nossa heurística na *Hotspot* JVM foi o fato desta não permitir a alteração, em tempo de execução, do tamanho da *heap*. Como nossa heurística exige a alteração do tamanho do Éden durante a execução da aplicação, procuramos alternativas para sua implementação na *Hotspot*

JVM. Nossa solução foi definir, no início da execução da *HotSpot* JVM, o tamanho máximo do Éden. A partir daí, o tamanho inicialmente disponibilizado para a aplicação foi definido como metade deste tamanho máximo. O Éden, neste caso, pode expandir-se até este tamanho máximo previamente definido.

O processo de aumento e redução do Éden é feito na classe *EdenSpace* do *DefNewGeneration*. Em sua implementação original, esta classe contém o limite inferior e o superior do espaço delimitado para o Éden. Os campos *bottom*, *top* e *end* (figura 3) 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*.

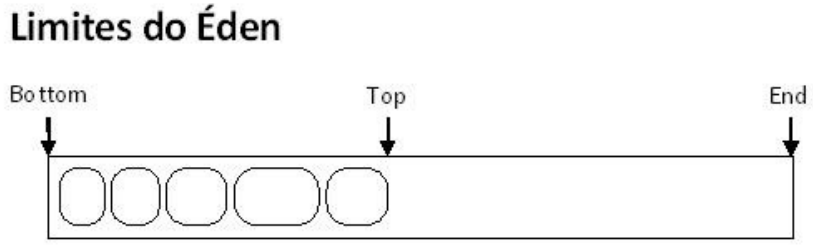

Figura 3. Limites do Éden

Para que o ajuste de tamanho do Éden funcionasse, modificamos a classe *EdenSpace* para criar um novo campo chamado *hardEnd* (figura 4). Esse campo tem por finalidade armazenar o limite superior do Éden. Com isso, o valor do campo *end* pode variar entre o *hardEnd* (seu valor máximo), e a metade do espaço delimitado por *bottom* e *hardEnd* (seu valor mínimo). Na inicialização do *EdenSpace*, os valores dos campos *end* e *hardEnd* são definidos como a figura 4 ilustra, sendo o valor do *hardEnd* igual ao final do espaço e o valor do *end*, o meio do espaço.

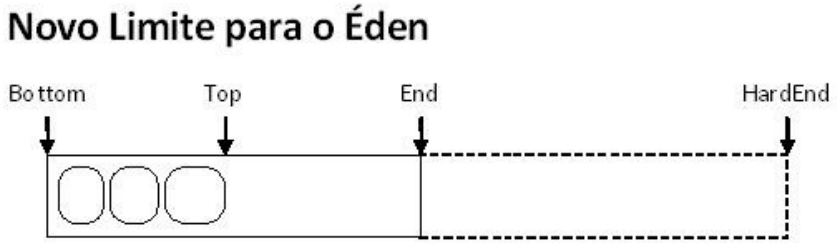

Figura 4. Novo Limite para o Éden

A alteração do tamanho do Éden só é feita após a execução de uma coleta de lixo, já 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.

Em nossa primeira tentativa de implementação, definimos que o Éden se expande a uma taxa igual à metade do espaço total ainda 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. Deve-se salientar que essa alteração só ocorre caso muitas coletas sejam executadas em um intervalo de tempo relativamente curto.

Definimos também que a taxa de redução do Éden é igual ao espaço ainda disponível para expansão, ou seja:

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

A fórmula acima é válida caso o espaço ainda disponível para alocação seja menor que a metade do espaço total do Éden. Caso contrário, a operação de redução tornaria o espaço para alocação igual à zero, o que causaria uma falha na próxima tentativa de alocação de memória na JVM.

Devemos também chamar a atenção para outro ponto ligado à redução do Éden: como definimos que a sua taxa de redução é igual ao espaço ainda disponível para expansão, não teríamos qualquer redução caso não existisse mais espaço disponível para o aumento do Éden. Neste caso, definimos que a redução será igual à 1/4 de seu tamanho total, ou seja, 25% de (*hardEnd – bottom*).

O pseudo-código abaixo ilustra a implementação de nossa heurística na JVM. Ele é sempre chamado após as operações de coleta de lixo, já que nesse ponto da execução o Éden não contém qualquer objeto, assim não há problemas na alteração de seu tamanho.

```
ColetaDados();
SE (NumeroColetas >= 30) E (TempoEntreColetas <= 300) ENTÃO
      End := ARREDONDA((HardEnd + End)/2);
SENÃO
      SE (NumeroColetas < 30) E (TempoEntreColetas > 300) ENTÃO
            SE End = HardEnd ENTÃO
                  End := End - ARREDONDA((HardEnd – Bottom)/4);
            SENÃO
                  End := 2*End – HardEnd;
```

Devemos salientar que o método ColetaDados(), como o próprio nome indica, coleta os dados necessário para a realização dos testes da heurística, como a quantidade de coletas realizadas e o tempo decorrido entre elas.

5. Resultados Preliminares

Utilizamos um conjunto de aplicativos com o propósito de avaliar o desempenho da heurística implementada no coletor serial da geração jovem. Os aplicativos são: GCOld [Detlefs 2005], GCBench [Boehm], JGFCreateBench, JGFSerialBench, e JGFSORBench, estes últimos parte do *suite* de aplicativos do Java Grande Fórum [Bull *et alli* 2000].

GCOld é um *benchmark* sintético que modela aplicações onde os dados mais antigos são mais propensos a se tornarem lixo. GCBench também é um *benchmark* sintético que divide a sua execução em duas fases distintas: a) a expansão da *heap* e b) a alocação e liberação de grandes porções de memória. JGFCreateBench aloca, durante a sua execução, inúmeros tipos de objetos, sem utilizá-los em processamentos posteriores. Como resultado, os objetos alocados têm uma vida extremamente curta, não

sobrevivendo a uma primeira limpeza da memória. JGFSerialBench realiza operações de serialização em diversos objetos, ou seja, armazena em disco o estado dos objetos, recuperando-os posteriormente. São construídas durante a sua execução estruturas como árvores, *LinkedLists*, *Arrays* e *Vectors* que são gravados em arquivos e depois carregados novamente em memória. Para cada estrutura são realizadas 4000 iterações de gravação e leitura de arquivos. O consumo de memória nesta aplicação é aproximadamente constante para cada tipo de estrutura de dados. JGFSORBench realiza centenas de iterações do algoritmo *Successive Over Relaxation* em uma matriz de tamanho NxN. A característica básica deste algoritmo, em termos de uso de memória, é realizar toda a alocação de objetos no início do processamento.

Os testes foram realizados em uma máquina com processador AMD Sempron 2800+, 32 bits, com *clock* de 1.6 Ghz, 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 de seu código fonte e a outra com a implementação da heurística. 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, a taxa de *throughput* e o *footprint*. O *throughput* representa nesse contexto o percentual de tempo que a JVM não executa coleta de lixo, tendo como referência o tempo total de execução do aplicativo. O *footprint* é o uso máximo de memória do aplicativo.

Para avaliar os resultados obtidos, geramos o *log* da coleta de lixo para cada uma das aplicações. O aplicativo *GCViewer* [Tagtraum industries 2005] foi utilizado para visualizar estes arquivos.

Cada aplicativo foi executado no mínimo três vezes de forma exclusiva, e a partir dos resultados foi computada uma média dos valores obtidos. O desvio padrão dos tempos de execução foi desprezível.

A JVM tem duas versões distintas: cliente e servidor. Essencialmente tratam-se de dois compiladores distintos que possuem uma interface comum com o sistema de execução da JVM. A versão cliente tem seu compilador ajustado para executar aplicações que necessitam de a) um tempo de iniciação rápido ou b) pequenos *footprints*. Já a versão servidor tem o compilador ajustado para aplicações onde o desempenho global é o ponto mais importante. Nosso trabalho avaliou o impacto de nossa heurística nas duas versões, mas, por falta de espaço, neste trabalho reportamos apenas os resultados com a versão cliente.

Os testes foram realizados usando duas máquinas Java distintas. A primeira máquina é a *HotSpot* JVM padrão, executando em sua versão cliente, com tamanho total da *heap* igual à 256 MB. Para esta máquina padrão variamos o tamanho da geração jovem de três formas distintas: 1) nesta configuração a geração jovem tem um tamanho fixo igual à 32 MB, 2) nesta configuração seu tamanho varia entre 32 MB e 64 MB, e 3) nesta configuração seu tamanho é fixo em 64 MB. Por fim, a segunda máquina Java consiste da *HotSpot* JVM modificada pela implementação de nossa heurística (4), com *heap* de 256 MB e a geração jovem variando entre 32 MB e 64 MB .

A diferença entre nossa heurística e a implementação da JVM que expande a geração jovem (versão 2) é a forma como as expansões se dão. Na implementação nativa da JVM, quando não há mais espaço disponível na geração jovem, a JVM tenta primeiro expandir o seu tamanho até que o limite especificado seja atingido. Caso seja possível realizar a expansão, a alocação é feita sem que sejam necessárias coletas de lixo. Caso não seja possível, ou seja, caso o limite superior já tenha sido atingido em operações de expansão anteriores, realiza-se a coleta de lixo para liberar espaço nesta geração. Logo, na versão 2 avaliada, a expansão da memória é realizada antes que ocorra uma coleta. No caso da nossa heurística, várias coletas ocorrem antes que o tamanho da geração seja modificado. Nossa heurística utiliza justamente as informações das coletas para saber se estão ocorrendo muitas alocações, optando pelo aumento da geração jovem apenas quando este quadro se confirma, enquanto que na implementação nativa da JVM o objetivo é evitar que ocorra uma coleta. Além do mais, podemos reduzir, em tempo de execução, o tamanho da geração quando o espaço adicional de memória não mais se faz necessário.

Para a JVM alterada com a heurística proposta neste trabalho, utilizamos a política de aumentar o tamanho do Éden após 30 coletas menores seguidas executadas em intervalo de tempo de 100 milissegundos. A redução do tamanho do Éden se dá sempre que o tempo entre duas coletas consecutivas torna-se relativamente grande; no caso deste trabalho isto ocorre quando o intervalo entre duas coletas torna-se superior à 3 segundos.

Os resultados obtidos são apresentados em tabelas, onde as linhas representam as quatro configurações testadas e as colunas apresentam os resultados das métricas avaliadas. As configurações 1 a 3 foram executadas na máquina *HotSpot* JVM padrão, sem qualquer modificação, enquanto na versão 4 a máquina *HotSpot* JVM foi modificada pela implementação de nossa heurística.

5.1. GCOld

Este aplicativo cria muitos objetos grandes que são mantidos alocados na memória durante muito tempo. Assim a geração antiga é muito utilizada, o que faz com que coletas maiores sejam realizadas com mais freqüência. Como nossa heurística tenta reduzir a quantidade de coletas menores, esperávamos não ter qualquer ganho neste *benchmark*. Neste caso, nosso objetivo primário foi verificar se algum *overhead* seria adicionado para aplicativos com esta característica de uso de memória.

Tabela 1 - Resultados do aplicativo GCOld

	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

Observando os resultados da tabela 1 percebemos que a heurística obteve uma taxa de *throughput* um pouco menor do que as obtidas por outras configurações.

Comparado com a configuração 3, a heurística conseguiu reduzir significantemente o desperdício de memória. Apesar da heurística ter obtido o pior tempo acumulado de coletas, o tempo médio de cada uma das pausas foi bem inferior ao da pior configuração e o tempo total da execução da aplicação ficou dentro da média das demais configurações.

5.2. 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.

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

Tabela 2 - Resultados do aplicativo GCBench

	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

A configuração 1 representa a pior execução, considerando o tempo como variável de comparação; do tempo total da execução do aplicativo aproximadamente 36% foram gastos com a execução do aplicativo, ou seja, o custo da execução dos coletores foi muito maior do que o da execução do aplicativo. A configuração 3, que executa o aplicativo com 64 MB para a geração jovem, obteve o melhor tempo de execução. Isso ocorreu devido a quantidade de coletas realizadas ser bem menor, o que tomou cerca de 26% do tempo total de execução.

Com relação à heurística, observamos uma redução no tempo total de execução do aplicativo em comparação às configurações 1 e 2, mas não tão bom quanto a obtida pela configuração 3. O aumento do tamanho do Éden, neste caso, foi responsável pela redução no número de coletas, o que em última instância reduz o tempo de execução do aplicativo. A versão da JVM com a heurística reduziu o tempo de execução do aplicativo em 20%, mas com um custo na utilização de memória: na tentativa de adaptar o tamanho do Éden foi necessário utilizar 12% a mais de memória do que as configurações 1 e 2.

5.3. 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 aos do GCBench.

Os resultados obtidos são apresentados na tabela 3. Pode-se perceber que, de forma geral, 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 de execução em 7%, mas o tempo acumulado gasto com coletas foi ligeiramente maior do que o das demais configurações. A média de pausa foi também ligeiramente pior. Esses dois resultados nos levam a concluir que a heurística conseguiu reduzir a quantidade total de coletas na geração jovem, o que levou a redução no tempo total de execução da aplicação, mesmo tendo uma taxa de *throughput* um pouco mais baixa. 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.

Tabela 3 – Resultados do aplicativo JGFCreateBench

	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

Outro ponto que pode ser observado 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 primário da heurística era reduzir a quantidade de coletas; o que acaba fazendo com que uma quantidade maior de lixo seja mantida na memória por mais tempo, o que não ocorre com a configuração 2.

5.4. JGFSerialBench

A heurística proposta obteve, para este aplicativo, ótimos resultados. A redução do tempo total de execução ficou cerca de 11% abaixo da média dos tempos das demais configurações. Isso aconteceu pelo fato deste aplicativo apresentar justamente as características que desejávamos atender com nossa heurística: alocações de grandes quantidades de memória de maneira relativamente constante ao longo de toda a execução.

Tabela 4 – Resultados do aplicativo JGFSerialBench

	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

Deve-se chamar a atenção para outro aspecto interessante: o uso total de memória (*footprint*) foi menor na heurística do que nas outras configurações. Isso ocorreu basicamente porque os objetos alocados tornaram-se lixo ainda na geração jovem, quando a heurística já havia disponibilizado todo o espaço possível para

alocação. Acreditamos que isso evitou operações de cópia destes objetos para outros espaços de memória, o que por fim reduziu o tempo médio de cada operação de coleta e o tempo total de execução do programa.

5.5. JGFSORBench

Esse aplicativo realiza muito processamento, alocando os dados necessários no início da execução. Espera-se que poucas coletas sejam realizadas e, assim como GCOld, nossa heurística não traga grandes ganhos para a sua execução. Esperamos então repetir os resultados de GCOld, não adicionando grandes *overheads* para a execução da aplicação.

A configuração 3 obteve os melhores resultados, mas a diferença em comparação as demais configurações é mínima. Como era desejável, em aplicativos que realizam poucas operações de coleta de lixo na geração jovem, o uso da heurística não adicionou *overheads*, o que demonstra que a heurística não possui grandes custos adicionais de implementação. De fato, podemos até observar pequenas vantagens em sua utilização, já que o *footprint* foi menor com a nossa heurística do que as configurações que utilizam a mesma quantidade de memória (32 MB).

Tabela 5 – Resultados do aplicativo JGFSORBench

	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

6. Trabalhos Futuros

Nosso trabalho encontra-se em fase inicial de desenvolvimento, de forma que temos várias propostas de trabalhos futuros. A primeira proposta consiste da avaliação dos parâmetros que utilizamos para expandir e reduzir o tamanho do Éden. Acreditamos que um ajuste nas taxas de expansão e de redução do tamanho da *heap*, bem como nos parâmetros que disparam estas ações, podem melhorar o desempenho da heurística. Uma outra proposta consiste de ajustar dinamicamente estes parâmetros conforme o comportamento da aplicação. Outra proposta consiste da aplicação de nossa heurística nas demais gerações do coletor, ou mesmo da criação de uma heurística específica para estas. Acreditamos que o emprego de uma heurística para a geração antiga, por exemplo, poderia impactar positivamente os resultados dos aplicativos que mantém dados alocados por um período longo de tempo, como é o caso do GCOld. Gostaríamos também de avaliar o uso de nossa heurística em outras aplicações, de forma a avaliar melhor o impacto de nossa heurística em suas execuções. Pretendemos também instrumentar a JVM para colher mais informações a respeito das coletas, de forma a termos mais informações para fazermos nossas avaliações de desempenho. Por fim, nossa última proposta consiste de um melhoramento nos algoritmos de coleta por cópia hoje implementados na *HotSpot* JVM. Gostaríamos de avaliar o impacto do algoritmo *Mark Lazy-Sweep* [Jones e Lins 1996] na geração jovem. Esse algoritmo consiste da

marcação dos objetos vivos, assim como ocorre no algoritmo de cópia. Entretanto, a cada alocação de novos objetos, esse algoritmo procura na *heap* por espaços não marcados que sejam suficientemente grandes para comportar os novos objetos. Assim, esse algoritmo reduz o tempo de pausa do coletor, transferindo seu custo para a fase de alocação.

7. Conclusão

Neste trabalho, propomos, implementamos e avaliamos na *HotSpot* JVM uma nova heurística para realizar automaticamente o ajuste do tamanho da memória de acordo com as características de consumo de memória do aplicativo em execução.

A heurística proposta verifica o tempo decorrido dentre certas quantidades de coletas de lixo. Dependendo do tempo decorrido, aumenta-se ou reduz-se a memória disponibilizada para armazenar os objetos recém-alocados. A heurística baseia-se no fato de que tempos curtos entre coletas apontam muitas alocações ocorrendo na aplicação. Aumentamos então a região de memória destinada a armazenar objetos recém alocados na tentativa de suprir essas alocações com mais memória, sem que sejam necessárias operações de coleta para disponibilizá-la. Analogamente, se o tempo entre as coletas for relativamente grande, concluímos que poucas alocações estão ocorrendo. Neste caso reduzimos o tamanho da memória com o intuito de economizá-la.

A heurística mostrou-se uma boa alternativa para aplicativos que, ao longo de sua execução, alocam grandes quantidades de objetos e os utilizam por pouco tempo, sem acarretar em aumento de custos para aplicativos com características distintas.

Referências

Lindholm, T.; Yellin, F. (1999) "*The Java Virtual Machine Specification*", 2nd Edition, Prentice Hall PTR.

Bacon, D.; Cheng, P.; Rajan, V. T. (2004) "*A Unified Theory of Garbage Collection*". In: ACM SIGPLAN Notices, Vol. 39, Issue 10, pp. 50-68. Nova Iorque: ACM Press.

Ungar, D. M. (1984) "*Generation scavenging: A non-disruptive high-performance storage reclamation algorithm*". In: ACM SIGPLAN Notices, vol. 19, issue 5, pp. 157-167, ACM, 1984.

Boehm, H. "An Artificial Garbage Collection Benchmark" http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html

Detlefs, D. (2005) "GCOld: a benchmark to stress old-generation collection", http://www.experimentalstuff.com/Technologies/GCold/index.html,id=7phrthka4sqps132dpihr2?template.fileName=admin/printing.template

Bull, J. M.; Smith, L. A.; Westhead, M. D.; Henty, D. S.; Davey, R. A.. (2000) "A Benchmark Suite for High Performance Java", In: Concurrency: Practice and Experience, vol. 12, pp. 375-388, 2000.

Tagtraum industries, inc (2005) http://www.tagtraum.com.

Jones, Richard e Lins, Rafael. (1996). Garbage Collection: Algorithms for automatic dynamic memory management. Publicado por John Wiley & Sons. 329p.