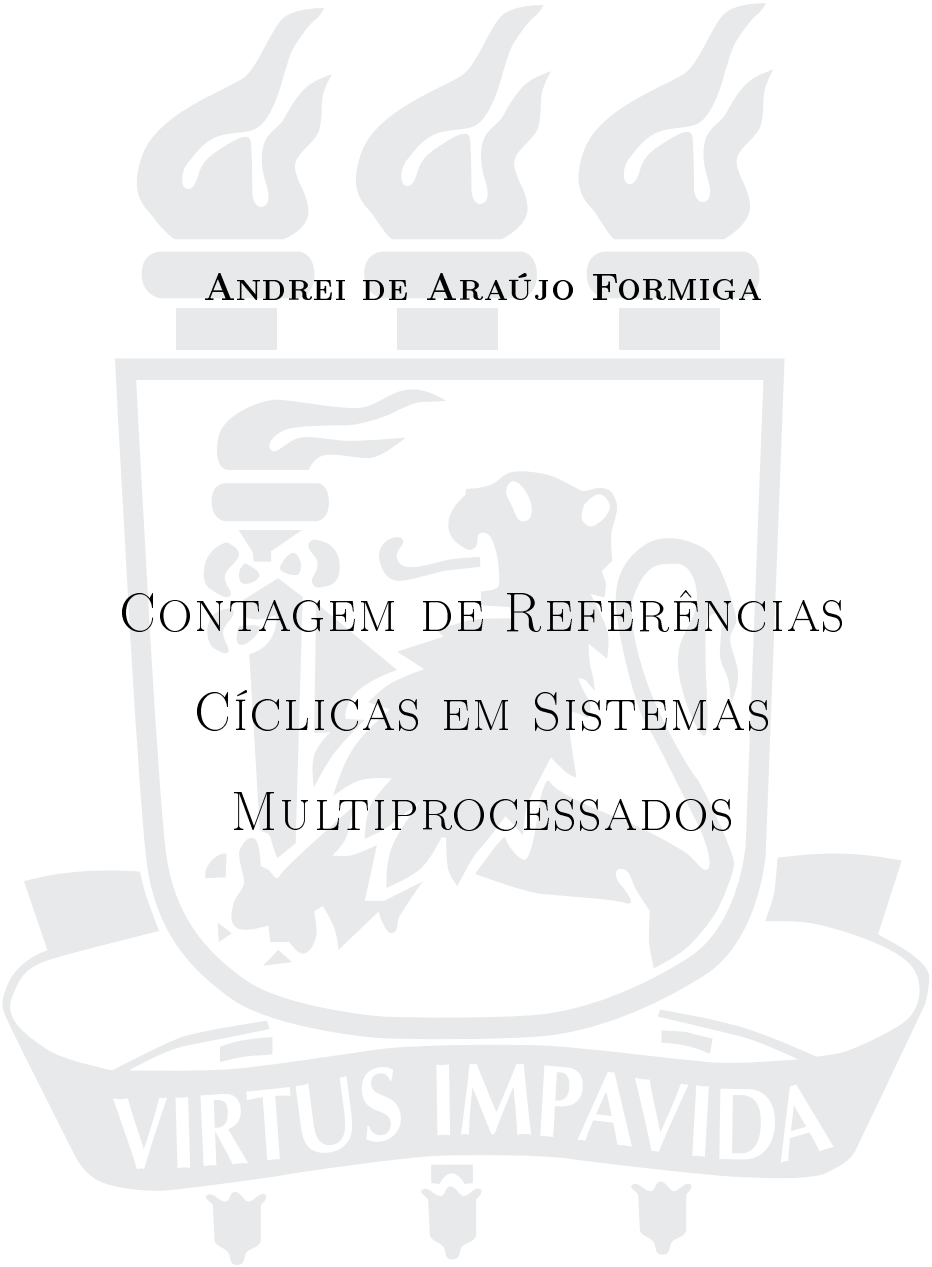
Universidade Federal de Pernambuco
Centro de Tecnologia e Geociências
Programa de Pós-graduação em Engenharia Elétrica



Andrei de Araújo Formiga

Contagem de Referências Cíclicas em Sistemas Multiprocessados

Recife, Outubro de 2006.

Andrei de Araújo Formiga

Contagem de Referências Cíclicas em Sistemas Multiprocessados

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para obtenção do grau de **Mestre em Engenharia Elétrica**

Orientador: Prof. Rafael Dueire Lins, Ph.D.

Recife, Outubro de 2006.

.

F725c **Formiga, Andrei de Araújo**

Contagem de referências cíclicas em sistemas multiprocessados / Andrei de Araújo Formiga. – Recife: O Autor, 2006.

93 f.; il., gráfs., tabs.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Elétrica, 2006.

Inclui Referências Bibliográficas e Apêndices.

1. Engenharia elétrica. 2. Contagem de Referências. 3. Gerenciamento de Memória. 4. Dependências Cíclicas. 5. Concorrência. 6. Paralelismo. I. Título.

UFPE

621.3 CDD (22.ed.) BCTG/2010-032

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE TESE DE MESTRADO ACADÊMICO DE

ANDREI DE ARAÚJO FORMIGA

TÍTULO

"CONTAGEM DE REFERÊNCIAS CÍCLICAS EM SISTEMAS MULTIPROCESSADOS"

A comissão examinadora composta pelos professores: RAFAEL DUEIRE LINS, DES/UFPE, VALDEMAR CARDOSO DA ROCHA JÚNIOR, DES/UFPE e FRANCISCO HERON DE CARVALHO JÚNIOR, DC/UFC, sob a presidência do primeiro, consideram o candidato **ANDREI DE ARAÚJO FORMIGA** **APROVADO**.

Recife, 20 de outubro de 2006.

JOAQUIM FERREIRA MARTINS FILHO
Coordenador do PPGEE

RAFAEL DUEIRE LINS
Orientador e Membro Titular Interno

FRANCISCO HERON DE CARVALHO JÚNIOR
Membro Titular Externo

VALDEMAR CARDOSO DA ROCHA JÚNIOR
Membro Titular Interno

Agradecimentos

Qualquer pessoa que tenha passado pelo processo de preparar uma dissertação sabe que esse é um trabalho que não se faz sozinho; seria impossível. É inevitável que várias pessoas, através de suas contribuições diretas e indiretas, influenciem no resultado final – algumas delas ao envolver-se com o trabalho em questão, outras como parte da vida do autor. Este espaço de agradecimentos é uma tentativa de reconhecer ao menos as contribuições mais importantes.

Sendo assim, agradeço: ao professor Rafael Dueire Lins, por ter acreditado em mim e aceitado me orientar, pela sua dedicação e orientação neste trabalho de mestrado e na vida acadêmica; aos meus pais, que não só apoiaram minha decisão de fazer pós-graduação como possibilitaram minha educação e apoio durante todos esses anos, e ainda incentivaram meus estudos e a vontade de aprender mais; aos colegas de mestrado e estudo, especialmente André Ricardson, Bruno Ávila e Márcio Lima, tanto pela convivência e amizade nesses meses quanto pela ajuda concreta; aos professores com quem tive contato no DES: Valdemar Cardoso da Rocha Jr., Hélio Magalhães de Oliveira, Ricardo Campello e Cecílio Pimentel, pelo exemplo estabelecido de bons pesquisadores e bons professores, um modelo a ser seguido pelos estudantes; finalmente, sem o apoio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico, CNPq, este trabalho não teria sido possível. A todos, muito obrigado.

Andrei de Araújo Formiga

Universidade Federal de Pernambuco

20 de Outubro de 2006

Resumo da Dissertação apresentada à UFPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Engenharia Elétrica

Contagem de Referências Cíclicas em Sistemas Multiprocessados

Andrei de Araújo Formiga

Outubro/2006

Orientador: Prof. Rafael Dueire Lins, Ph.D.
Área de Concentração: Comunicações
Palavras-chaves: contagem de referências, gerenciamento de memória, dependências cíclicas, concorrência, paralelismo.
Número de páginas: 93

O gerenciamento automático da memória utilizada por um programa se tornou uma necessidade na maioria das linguagens de programação em uso atualmente, cujas implementações incluem em sua maioria um coletor de lixo para administrar a memória utilizada. Dentre as técnicas para realizar o gerenciamento automático da memória, a contagem de referências se mostra popular por uma série de vantagens, dentre elas o fato de ser uma técnica naturalmente incremental, o que evita a parada completa da computação para realizar tarefas de administração da memória. Esta natureza incremental da contagem de referências indica que o algoritmo pode ser adaptado para uma versão concorrente, em sistemas multiprocessados, com facilidade. Entretanto, o problema da sincronização pode anular os ganhos de eficiência obtidos com essa extensão, inviabilizando o uso de um coletor de lixo concorrente. Nesta dissertação apresenta-se uma nova arquitetura para implementar concorrentemente o gerenciamento automático da memória baseado na contagem de referências; esta proposta está baseada em versões anteriores dos algoritmos seqüencial e concorrente para contagem de referências cíclicas, mas tem como diferencial o uso de sincronização mais eficiente. Os resultados de testes realizados com uma implementação desta nova arquitetura indicam que, de fato, a eficiência obtida compensa o seu uso em sistemas multiprocessados.

Abstract of Dissertation presented to UFPE as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering

Multiprocessing Cyclic Reference Counting

Andrei de Araújo Formiga

October/2006

Supervisor: Prof. Rafael Dueire Lins, Ph.D.
Area of Concentration: Communications
Keywords: reference counting, memory management, cyclic dependencies, concurrency.
Number of pages: 93

Automatic memory management has become a requirement in most current programming languages; it is expected that most implementations will include a garbage collector to manage memory. One of the well-known techniques for managing memory is reference counting, which has become popular for various reasons, one of them being its intrinsic incremental nature, in contrast to tracing collectors that stop all useful computation to complete its management tasks. Being an incremental algorithm, reference counting seems easy to extend to concurrent and parallel versions, adequate to multiprocessor systems. However, synchronization problems have the potential to compensate for any performance gains attained through concurrency, making the use of a concurrent collector impractical. Here it is presented a new architecture for concurrent reference counting; this proposal is based both on previous versions of the sequential reference counting algoritm and in previous concurrent architectures, but has as a distinctive advantage having lower synchronization requirements, attaining a more efficient algorithm. Performance tests executed on an implementation of this new architecture indicate that it is adequate for use in multiprocessor systems, presenting significant performance gains over the sequential version.

Sumário

Lista de Figuras

capítulo 1

Introdução

O objeto de estudo desta dissertação é o *gerenciamento automático e concorrente da memória*, utilizando algoritmos *concorrentes* para a *contagem de referências cíclicas*. Esta introdução apresenta uma explicação expandida sobre estes conceitos, os objetivos específicos do estudo, a motivação para pesquisar estes tópicos, e como a dissertação foi organizada para apresentar os resultados do estudo realizado.

1.1 Gerenciamento automático da memória

Qualquer pessoa que já tenha realizado cálculos não-triviais usando lápis e papel sabe que é preciso guardar valores intermediários que servirão para chegar aos resultados finais do cálculo. Nas máquinas de calcular atuais, os computadores, não é diferente: é preciso um espaço de armazenamento para guardar os dados, uma *memória*. Os computadores implementam hoje os modelos abstratos de máquinas de calcular que foram propostos ainda na primeira metade do século XX por Alan Turing e outros; no modelo da máquina de Turing já estava prevista uma memória, representada por uma fita que a máquina pode ler e escrever [20]. Entretanto, enquanto no modelo abstrato da máquina de Turing a fita é infinita – ou seja, o espaço para armazenamento é ilimitado – nas máquinas reais esse espaço é limitado pela realidade física. A memória é, portanto, um dos recursos essenciais e limitados dos computadores que devem ser gerenciados para que seu uso seja eficiente.

Em computadores atuais é comum ter vários programas sendo executados simultaneamente. Um aspecto do gerenciamento da memória é garantir que cada programa em execução

tenha acesso à memória do computador; isso é feito pelos sistemas operacionais e não é objeto de estudo aqui. Esta dissertação se ocupa do gerenciamento da memória em apenas um programa, de forma isolada de outros que possam ser executados simultaneamente com ele. Nos sistemas atuais esse enfoque é justificável, já que o sistema operacional cuida para que cada programa tenha acesso à memória como se ele fosse o único em execução.

É possível dar um exemplo da necessidade de um programa gerenciar sua memória voltando à analogia com uma pessoa realizando um cálculo com lápis e papel: imaginando um cálculo realmente complexo, à medida que os resultados intermediários são armazenados na folha de papel, o espaço disponível na folha será reduzido até que, eventualmente, seja necessário usar uma nova folha de papel para continuar os cálculos. Mas há uma outra alternativa: o calculador humano pode observar que resultados intermediários não são mais necessários, porque nenhum cálculo atual depende deles, e apagar com uma borracha o espaço ocupado por eles para utilizar com novos resultados. Esses resultados não mais necessários apenas desperdiçam espaço de armazenamento, e são portanto *lixo*.

Em um programa no computador ocorre o mesmo: dados que foram utilizados previamente se tornam desnecessários, e o espaço ocupado por eles será desperdiçado caso o programa não identifique esses dados. Este processo de identificar que partes da memória não estão mais em uso pode ser feito manualmente pelo criador do programa, ou automaticamente por uma parte do programa normalmente chamada de *coletor de lixo*, sendo essa segunda alternativa a preferida atualmente em projetos realistas de engenharia de *software*. De fato, praticamente todas as linguagens de programação que se tornaram populares nos últimos cinco anos utiliza alguma forma de gerenciamento automático de memória [4, 18, 33, 46].

1.2 Coleta de lixo concorrente

O objetivo do coletor de lixo, como visto, é dividir a memória de um programa em duas classes: as partes que ainda estão sendo utilizadas, e as que podem ser reaproveitadas para outros fins. Em sistemas tradicionais uniprocessados, esta tarefa é normalmente feita intercaladamente com as operações do resto do programa, já que só há um processador para executar as instruções.

Embora a maioria dos computadores em uso sempre tenha sido baseada em um único processador, são realidade há bastante tempo os sistemas multiprocessados. Uma extensão natural da técnica de coleta de lixo tradicional para sistemas multiprocessados é fazer com

que o coletor seja executado concorrentemente com o resto do programa, e de fato isso vem sendo estudado há décadas. Um dos primeiros trabalhos sobre gerenciamento automático e concorrente de memória data de 1975 [42]. Em tempos recentes tem se tornado claro que mesmo os computadores de uso doméstico se tornarão multiprocessados em um futuro próximo (vide a seção 1.5), e a pesquisa em métodos de gerenciamento automático e concorrente de memória se torna cada vez mais importante.

1.3 Contagem de referências cíclicas

Como será detalhado no capítulo 2, existem três técnicas principais para gerenciamento automático da memória, sendo uma delas a contagem de referências. Para entender como funciona esta técnica, primeiro considera-se que os dados na memória estão organizados em *objetos*. Um objeto pode ser apenas um número utilizado em cálculos, ou uma estrutura mais complexa como um registro em uma agenda telefônica, que contem nome, endereço, telefone e outros dados. É comum que um objeto faça referência a outros objetos na memória; e, de fato, um objeto só pode ser considerado em uso se ele for atualmente referenciado por algum outro objeto acessível. A contagem de referências opera, então, simplesmente mantendo um contador, em cada objeto, de quantos outros objetos o referenciam. Assim, qualquer objeto em uso possui uma contagem de referências maior que zero, e se a contagem de um objeto chega a zero ele pode ser corretamente considerado como fora de uso, e reutilizado posteriormente.

A idéia é bastante simples e foi idealizada por G.E.Collins [10] no primeiro ano dos estudos com sistemas de gerenciamento automático de memória, visando evitar as pausas de processamento impostas pelo algoritmo de cópia. Entretanto, como notado por J.H.McBeth [34], esta técnica não consegue detectar lixo formado por objetos que se referenciam ciclicamente. Ou seja, se o objeto A referencia o objeto B, e o objeto B possui uma referência ao objeto A, mas nenhum dos dois é referenciado por algum outro objeto, eles não estão em uso, mas um sistema baseado em contagem de referências não poderá detectar isso já que tanto A quanto B possuem um contador de referências com valor maior que zero.

Para resolver este problema foram desenvolvidos uma série de algoritmos para contagem de referências cíclicas, sendo o primeiro deles publicado por Martinez, Wachenchauzer e Lins [32] em 1990. Embora resolvesse o problema, a técnica original neste artigo perdia a eficiência do algoritmo de contagem de referências. Trabalhos posteriores [25, 28] apresentaram uma série de otimizações ao algoritmo de 1990 que melhoram sua eficiência significantemente, ao

custo de uma maior complexidade de implementação.

Lins também propôs arquiteturas para realizar o algoritmo de contagem de referências cíclicas de maneira concorrente em sistemas multiprocessados com memória compartilhada [27, 29, 31]. Esta dissertação continua a investigação por este caminho, como explicado a seguir.

1.4 Objetivos

O objetivo do estudo relatado nesta dissertação é investigar uma nova versão concorrente do algoritmo de contagem de referências cíclicas para sistemas multiprocessados de memória compartilhada. O ponto de partida é a arquitetura mais recente proposta por Lins [31], que é alterada para obter uma versão com sincronização mais eficiente. A arquitetura resultante foi implementada a fim de observar, através de medições em situações reais, qual sua eficiência comparativa em relação à versão seqüencial do algoritmo.

1.5 Motivação

Desde os primeiros tempos da computação, a conexão de vários processadores em paralelo para realizar uma máquina com maior poder de processamento é vista como uma forma de obter mais eficiência do projeto de computadores. Por várias vezes os cientistas da computação previram que os computadores paralelos se tornariam a regra, e não uma exceção relegada a nichos de mercado, e em todas as vezes essas previsões não se cumpriram. Já em 1967 Gene Amdahl escreveu ([1], *apud* [40]):

> Por mais de uma década os analistas anunciam que a organização de um único computador alcançou seus limites e que avanços verdadeiramente significantes só podem ser feitos pela interconexão de uma multiplicidade de computadores de tal modo que permita solução cooperativa... Demonstrou-se a continuada validade do método de processador único...

No momento atual essas previsões retornam mais uma vez, talvez agora com a maior chance de se concretizar. Os maiores fabricantes de microprocessadores têm lançado no mercado um número de produtos que utilizam tecnologia de multiprocessadores, muitas vezes integrados em um único *microchip*, no que é conhecido como *processadores multi-núcleo* [38]. Com a integração em escala cada vez maior, a estratégia tradicional de aumentar a freqüência

de operação para aproveitar o maior número de transistores em uma única pastilha se mostrou ineficiente a partir de um certo limite, devido principalmente ao aumento da potência consumida. Tornou-se atrativo, então, integrar vários processadores em uma única pastilha, provendo maior poder de processamento sem aumentar a freqüência de operação. Esta é a estratégia adotada atualmente pela Intel e AMD, que lançaram em 2005 processadores com dois núcleos; a mesma tendência também tem como exemplos o processador Niagara da Sun e o Cell Processor da IBM e Sony. A Intel anunciou em 2004 a mudança em sua estratégia [5] e recentemente revelou que tem planos para um processador com 32 núcleos em 2010 [17].

Neste cenário, a expectativa é que nos próximos anos mesmo os computadores domésticos serão multiprocessados, restando uniprocessadores apenas em sistemas legados. A grande questão que se coloca hoje é quanto tempo o *software* demorará para acompanhar o *hardware*, já que a programação de sistemas concorrentes sempre foi menos estudada e menos compreendida que nos sistemas seqüenciais. A dificuldade para que os programadores utilizem adequadamente os recursos de um computador multi-núcleo tem chamado a atenção tanto na indústria de *software* quanto nos meios de pesquisa; comenta-se em algumas situações que uma revolução nas técnicas de programação está prestes a acontecer [43].

Um candidato óbvio para aproveitar o paralelismo disponível é o sistema de coleta de lixo. Este é um trabalho que é feito automaticamente, sem a intervenção explícita do programador, e que opera em funções que são ortogonais às do resto do programa. Aproveitar o paralelismo no gerenciamento de memória pode tornar qualquer programa automaticamente mais eficiente em arquiteturas paralelas, sem a necessidade de esforço por parte dos programadores. Trata-se então de uma contribuição imediata a esta situação de transição, ajudando os sistemas de software a aproveitar o poder computacional adicional disponível.

Qualquer sistema concorrente de gerenciamento automático de memória pode tirar vantagem dos novos processadores multi-núcleo, mas os métodos baseados na contagem de referências possuem algumas vantagens, como será visto no Capítulo 2.

1.6 Organização da dissertação

A dissertação está dividida da seguinte maneira.

O Capítulo 2 trata do gerenciamento de memória, incluindo um curto histórico e uma análise superficial das três principais técnicas de gerenciamento automático de memória; neste ponto se estabelecem as vantagens e desvantagens da contagem de referências com relação às

outras técnicas, sendo a sua principal falha a não-recuperação de estruturas cíclicas. A solução para este problema são os métodos para contagem de referências cíclicas, cuja versão mais recente é apresentada no Capítulo 3; o algoritmo neste capítulo está na sua forma seqüencial, para sistemas uniprocessados.

Em seguida, o Capítulo 4 trata dos algoritmos concorrentes para gerenciamento automático da memória. Primeiro são apresentadas idéias gerais sobre o problema e as dificuldades envolvidas. Em seguida, apresentam-se as versões da nova arquitetura proposta para contagem de referências em sistemas multiprocessados, incluindo todas as rotinas que fazem parte do algoritmo. Sendo uma área ativamente pesquisada, também são apresentados alguns trabalhos relacionados, comparando suas abordagens com a arquitetura de partida.

No Capítulo 5 são mostrados os resultados dos testes realizados com a implementação da arquitetura proposta. Os testes comparam o algoritmo concorrente com a versão seqüencial e com uma versão que não realiza nenhum gerenciamento automático da memória, e é portanto parcialmente incorreta, mas que serve para situar o impacto da inclusão do gerenciamento automático da memória sobre o desempenho dos programas usuários.

Finalmente, o Capítulo 6 conclui o trabalho com as considerações finais e sugestões para trabalhos futuros que continuem a pesquisa relatada aqui.

O trabalho ainda inclui dois apêndices; o primeiro descreve a implementação utilizada para os testes, e o segundo mostra os programas de teste utilizados.

capítulo 2

Gerenciamento de Memória

Desde os primeiros computadores automáticos, a memória é um dos principais recursos cujo uso deve ser gerenciado ou arbitrado em uma máquina computacional. A lei de Moore [37] prevê que o número de transistores em circuitos integrados (de tamanho e custo fixo) dobra a cada 18 meses, fato que garantiu que a quantidade de memória disponível nos computadores crescesse em progressão geométrica. E quanto mais memória ficava disponível, mais usos e aplicações foram encontradas para os computadores, e as aplicações foram se tornando mais complexas. Do aumento de complexidade das aplicações seguiram-se maiores exigências para o gerenciamento de memória, principalmente com relação à eficiência temporal. Os computadores tornaram-se multitarefa e multiusuário, com sistemas operacionais cada vez mais complexos. Para contrabalançar as maiores dificuldades impostas à tarefa de projetar e criar programas para os computadores atuais, o seu maior poder de processamento permite a automatização de tarefas que antes exigiam atenção específica do programador. O gerenciamento automático da memória é uma dessas tarefas, e de fato a experiência tem confirmado que os programadores se tornam mais produtivos, e os programas criados mais robustos, quando podem utilizar um sistema de gerenciamento automático de memória. Isso se reflete no fato da maioria das linguagens de programação surgidas nos últimos anos incluírem um gerenciador de memória que livra o programador do trabalho de gerenciamento manual.

Neste capítulo é apresentado um panorama geral sobre o gerenciamento de memória, enfocando principalmente as técnicas de gerenciamento automático e seqüencial – para com-

putadores uniprocessados. Primeiro, definem-se os conceitos necessários e a notação utilizada. Seguem-se então um breve histórico da evolução do gerenciamento de memória e uma descrição resumida das três principais técnicas para gerenciamento automático, finalizando com a versão clássica da contagem de referências, que é o tema desta dissertação. Quando são apresentadas as vantagens e desvantagens do algoritmo de contagem de referências, uma atenção especial é dada ao problema no tratamento de estruturas cíclicas; a solução é descrita no Capítulo 3.

2.1 Definições e Notação

A memória RAM (*Random Access Memory*) de um computador é um conjunto de *bytes* organizados de tal forma que cada *byte* individual está associado a um endereço único; o valor de cada *byte* pode ser lido ou alterado livremente, sendo necessário apenas conhecer seu endereço para isso.

O objetivo do gerenciamento da memória é diferenciar as partes da memória que estão em uso – ou seja, *ativas* – das partes que não estão ativas. Estas últimas podem ser reaproveitadas para outros usos. As partes não utilizadas pelo programa mas que ainda estão marcadas como ocupadas são denominadas de *lixo*. O subsistema do programa (ou sistema de tempo de execução) que realiza o gerenciamento automático da memória, reaproveitando partes não utilizadas, é comumente conhecido pelo nome de *coletor de lixo*, e o próprio processo de gerenciamento é chamado de *coleta de lixo*. Quando algumas partes da memória não utilizada não são identificadas como tal, e ficam inacessíveis para o programa pelo resto do seu tempo de vida, diz-se que ocorreu um *vazamento de memória* (*space leak*).

Mais interessante para o gerenciamento de memória no contexto das linguagens de programação é interpretar a memória como ocupada por *objetos* de algum programa. Um objeto é uma estrutura de dados que ocupa uma região definida da memória, ou seja, um conjunto definido de *bytes*. Uma *variável* é uma entidade do programa usada para fazer referência a um objeto. Objetos podem ser *atômicos*, caso não possam ser analisados e separados em partes, ou *compostos* (também chamados de *estruturados*) caso sejam formados por composição de outros objetos; estes componentes, chamados de *campos*, podem ser atômicos ou compostos. Um tipo de objeto atômico particular é um *ponteiro* (também chamado de *referência*). No modelo considerado aqui, um ponteiro contém como valor o endereço de algum outro objeto na memória; se o ponteiro P possui como valor o endereço do objeto O, diz-se que P *aponta*

para O. Um valor especial para ponteiros, chamado de *nil*, é usado quando nenhum objeto é apontado, ou seja, P tem valor *nil* quando não existe um objeto O tal que P aponta para O. Um ponteiro pode fazer parte de algum objeto composto, e assim diz-se que um objeto O_1 aponta para um outro objeto O_2 quando O_1 tem como componente um ponteiro para O_2. Pode-se dizer também neste caso que O_1 referencia O_2.

Essa organização da memória em objetos que podem apontar para outros objetos é representada por um grafo direcionado no qual os objetos são os nós e existe uma aresta de O_1 para O_2 se O_1 aponta para O_2. Este grafo é chamado de *grafo da memória* e captura o estado da memória em algum momento, e apenas nesse momento.

Uma parte ativa da memória é aquela que está ocupada por algum objeto em uso pelo programa. Um objeto O só pode estar em uso se ele for referenciado por algum outro objeto que esteja em uso, ou seja, se algum objeto ativo O' contém um ponteiro para O. Obviamente, esta cadeia de ponteiros deve começar em algum lugar, e este lugar é o chamado *conjunto raiz*[1] – um conjunto de objetos a partir do qual, seguindo-se as referências em ponteiros, pode-se chegar a qualquer objeto ativo no programa.

Achar os objetos ativos, então, corresponde a determinar que nós no grafo da memória estão transitivamente conectados a algum nó do conjunto raiz. Os nós que não estiverem conectados ao conjunto raiz são lixo e podem ser reaproveitados. Todas as técnicas criadas para o gerenciamento da memória realizam este processo, de alguma forma.

2.2 Histórico

Aqui é apresentado um breve histórico sobre a evolução das técnicas de gerenciamento de memória. Este resumo histórico tem apenas a intenção de explicitar os motivos por trás do desenvolvimento e uso das técnicas de gerenciamento automático. O conteúdo desta seção foi baseado no livro de Jones e Lins [21], onde o leitor encontrará maiores detalhes sobre o assunto.

Na história do desenvolvimento das linguagens de programação repetem-se casos do seguinte padrão:

1. Uma tarefa de programação é inicialmente feita manualmente;
2. Técnicas para automatizar a tarefa são desenvolvidas, mas julgadas computacionalmente ineficientes para que seu uso se torne comum. Os programadores julgam que o custo

[1] Em linguagens de programação o conjunto raiz é formado por ponteiros dos registros de ativação

computacional de usar as técnicas automáticas não compensam a maior produtividade ganha com elas;

3. Com o avanço no projeto de semicondutores, aumenta-se tanto o poder computacional quanto a memória disponível para os computadores, o que altera a relação de custo e benefício para as técnicas automáticas. Eventualmente, alguma tarefa que anteriormente era feita de maneira manual ou explícita é automatizada porque julga-se que os custos se tornaram baixos o suficiente.

O gerenciamento da memória seguiu esse padrão: inicialmente feito apenas manualmente, técnicas para automatizá-lo logo surgiram, mas permaneceram ignorados por décadas por grande parte dos programadores, que julgavam ineficientes os métodos automáticos. Com o tempo, a relação entre custo e benefício mudou e hoje a maior parte dos programadores utilizam algum sistema que inclui o gerenciamento automático de memória. As etapas mais importantes nesta evolução são resumidamente contadas aqui.

Nos primeiros tempos da computação toda a comunicação entre programador e máquina era bit-a-bit, com chaves simples para entrada e LEDs para saída. Pouco tempo depois, a introdução de dispositivos simples de entrada e saída tornou mais simples a troca de valores hexadecimais entre operador e máquina. O próximo passo foi permitir que os programadores usassem códigos mnemônicos que eram mecanicamente traduzidos para a notação binária esperada pelo computador. Ainda assim, os usuários eram responsáveis por cada detalhe da execução de seus programas. Por exemplo, uma atenção especial era necessária para contar o número de palavras de memória usadas pelo programa e encontrar o endereço absoluto das instruções para determinar se havia espaço disponível para carregar o programa e para especificar o endereço de destino em instruções de desvio.

Pelo final dos anos 40 e início da década de 50, essa tarefa de gerenciamento foi transferida para os códigos macro e as linguagens de montagem (*assembly*). Programas simbólicos são mais fáceis de escrever e entender que programas de linguagem de máquina, primariamente porque códigos numéricos para endereços e operadores são substituídos por códigos simbólicos que carregam mais significado para um usuário. Ainda assim o usuário precisava se preocupar intimamente com a forma que um computador específico operava, e como e onde os dados eram representados dentro da máquina. O grande número de pequenos detalhes específicos da máquina que devem ser cuidados torna, até hoje, a programação em linguagem de montagem uma tarefa extenuante.

Para vencer estes problemas, idéias para linguagens de programação de alto-nível apareceram da metade para o fim da década de 40, com o objetivo de tornar a programação mais simples. Em 1952 apareceram os primeiros compiladores experimentais, e o primeiro compilador FORTRAN foi lançado no início de 1957. Um compilador para uma linguagem de programação de alto-nível deve alocar recursos da máquina alvo para representar os objetos de dados manipulados pelo programa do usuário. Existem básicamente três formas para alocar memória em computadores:

- ▷ alocação estática;
- ▷ alocação na pilha;
- ▷ alocação dinâmica (ou no *heap*).

Cada uma delas é detalhada nas subseções seguintes.

2.2.1 Alocação estática

A política de alocação mais simples é a de alocação estática. Todos os nomes no programa são associados a localizações na memória em tempo de compilação: estas associações não mudam em tempo de execução. Isto implica que as variáveis locais de um procedimento são associadas às mesmas localizações em cada ativação do procedimento. Alocação estática foi a política originalmente usada nas implementações da linguagem FORTRAN, e ainda foi utilizada na linguagem Fortran 77 e a linguagem de programação paralela Occam, por exemplo. A alocação estática possui três limitações:

- ▷ o tamanho de cada estrutura de dados deve ser conhecida em tempo de compilação;
- ▷ nenhum procedimento pode ser recursivo já que todas as suas ativações compartilham as mesmas localizações para as variáveis locais;
- ▷ estruturas de dados não podem ser criadas dinamicamente.

Apesar destes problemas, a alocação estática tem duas vantagens importantes: eficiência e segurança. Implementações de linguagems que utilizam alocação estática são geralmente rápidas pois nenhuma estrutura de dados (por exemplo pilhas) precisam ser criadas ou destruídas durante a execução do programa. Como todas as localizações de memória são conhecidas durante a compilação, os dados podem ser acessados diretamente ao invés de indiretamente por ponteiros. A outra vantagem está relacionada à segurança do programa: nenhuma falha

pode ocorrer por falta de memória durante a execução, já que os requerimentos de memória são conhecidos antecipadamente.

2.2.2 Alocação na pilha

As primeiras linguagens estruturadas em blocos apareceram em 1958 com Algol-58 e Atlas Autocode. As linguagens estruturadas em blocos eliminam algumas dificuldades da alocação estática reservando espaço em uma pilha. Um *registro de ativação* é empilhado na pilha do sistema cada vez que um procedimento é chamado, e desempilhado quando o procedimento retorna. Como a seqüencia de chamadas de procedimentos em um programa em execução segue uma estrutura similar a um percurso em profundidade em uma árvore de chamadas, a estrutura dos registros de ativação na pilha é ideal para guardar, para cada procedimento, os objetos que são locais a ele no registro de ativação correspondente. A organização em pilhas tem cinco conseqüências:

- ativações diferentes de um procedimento não compartilham as mesmas localizações de memória para as variáveis locais. Chamadas recursivas são possíveis, desta forma aumentando significantemente o poder expressivo da linguagem;
- o tamanho das estruturas de dados locais como vetores podem depender de parâmetros passados para o procedimento;
- os valores de variáveis locais alocados na pilha não podem persistir de uma ativação para outra do mesmo procedimento;
- o registro de ativação de um procedimento chamado não pode estar ativo por mais tempo que o registro do procedimento chamador;
- apenas um objeto cujo tamanho é conhecido em tempo de compilação pode ser retornado como o resultado de um procedimento.

2.2.3 Alocação dinâmica

Embora as linguagens estruturadas em blocos tenham resolvido muitos dos problemas existentes com a alocação estática, ainda assim era impossível criar estruturas de dados com tempo de vida arbitrário e independente de procedimentos específicos do programa. Para resolver isso, algumas linguagens como C e Pascal introduziram a possibilidade de alocar dados dinamicamente em uma estrutura conhecida como *heap*. Diferente da disciplina "o último a

entrar é o primeiro a sair" de uma pilha, dados presentes em um *heap* podem ser alocados e desalocados em qualquer ordem. A alocação dinâmica possui uma série de vantagens:

- ▷ o projeto de estruturas de dados pode incluir naturalmente estruturas recursivas como listas e árvores, dando-as representações concretas;
- ▷ o tamanho das estruturas de dados não precisa ser fixo, podendo variar dinamicamente. Exceder os limites pré-estabelecidos de estruturas de dados como vetores é uma das causas mais comuns de falhas em programas;
- ▷ objetos de tamanho dimensionado dinamicamente podem ser retornados por procedimentos;
- ▷ muitas linguagens de programação modernas permitem que procedimentos sejam retornados como resultado de outros procedimentos. Linguagens que usam alocação na pilha podem fazer isso se proibirem procedimentos aninhados: o endereço estático do procedimento retornado é usado (esta é a abordagem por trás dos ponteiros para função na linguagem C, por exemplo). Linguagens de programação funcional e de alta ordem podem permitir que o resultado de uma função seja uma *suspensão* ou *fechamento*: uma função armazenada juntamente com um *ambiente* que especifica associações de nomes com localizações de memória. Estas associações, portanto, têm um tempo de vida independente da função que as criou.

Atualmente muitas se não todas as linguagens de alto nível permitem a alocação dinâmica de memória tanto na pilha quanto no *heap*. Muitas linguagens tradicionais, como C e Pascal, deixaram a tarefa de gerenciar os dados alocados dinamicamente para o programador; ele deve especificar explicitamente quando uma região de memória alocada no *heap* pode ser liberada. A linguagem C++ seguiu este mesmo caminho, para manter a compatibilidade com a linguagem C. Já as linguagens funcionais, lógicas e a maioria das linguagens orientadas a objetos utilizam o gerenciamento automático de memória. Exemplos de linguagens com gerenciamento automático incluem Scheme, Dylan, Standard ML, Objective Caml, Haskell, Miranda, Prolog, Smalltalk, Eiffel, Java, Ruby, Python e Oberon. Algumas outras linguagens, como Modula-3, oferecem a possibilidade de gerenciamento automático ou manual.

A alocação manual de dados no *heap* pode ocasionar uma série de problemas relacionados ao tempo de vida dos dados. Um dos problemas é deixar de liberar uma região de memória que não é mais utilizada; outro é continuar acessando uma região que já foi liberada e pode estar sendo ocupada por dados diferentes. Além disso, a própria estrutura do programa se

torna mais complexa e difícil de manter pela necessidade de sempre controlar o tempo de vida das estruturas de dados. É um fato aceito atualmente na comunidade de desenvolvimento de *software* que o gerenciamento manual da memória deve ser utilizado apenas em um número limitado de casos, como por exemplo quando o controle total sobre a memória é realmente requerido, ou quando é necessário obter um desempenho específico para o programa que não pode ser obtido com gerenciamento automático.

2.3 Técnicas de gerenciamento automático de memória

Como visto na seção anterior, a administração do tempo de vida dos objetos alocados dinamicamente é o maior problema no gerenciamento de memória. Objetos alocados estaticamente ficam ativos durante toda a execução do programa, e não há preocupação em reaproveitar o espaço ocupado por eles; para os objetos alocados na pilha, seu tempo de vida é determinado pelo tempo de execução de algum procedimento, o que permite ao compilador gerar código para liberar o espaço ocupado por tais objetos quando eles não forem mais necessários – pois o procedimento que os contêm terminou sua execução.

O problema então é saber quando liberar o espaço dos objetos alocados dinamicamente. Em linguagens com gerenciamento manual da memória, o programador especifica explicitamente quando o espaço de um objeto deve ser liberado. Por exemplo, na linguagem C utiliza-se a função `free`. Para automatizar essa tarefa, é necessário identificar que objetos ainda estão em uso, e portanto acessíveis através do conjunto raiz, e quais não são mais necessários e podem ter seu espaço liberado. Ao longo da história do desenvolvimento do gerenciamento automático de memória surgiram três técnicas principais, cujas variantes são utilizadas até hoje, seja de forma direta ou misturadas em sistemas híbridos. São elas:

- marcar-e-varrer (*mark & sweep*);
- coleta por cópia;
- contagem de referências.

As subseções seguintes se ocupam da descrição destas três técnicas, incluindo especificações do seu funcionamento e comparativos de vantagens e desvantagens relativas de cada uma. Para simplificar a descrição, assume-se que os objetos no *heap* são todos de mesmo tamanho; tais objetos uniformes podem ser chamados de *células*. Com esta organização, as células livres do *heap* podem ser convenientemente organizadas em uma lista encadeada, chamada de *lista*

livre (*free list*). Admite-se que existem duas subrotinas simples disponíveis para o sistema de gerenciamento de memória: `allocate` retorna uma nova célula retirada da lista livre, e `free` inclui a célula passada como parâmetro para a lista livre, liberando-a para reutilização.

Os algoritmos usados para ilustrar e especificar a implementação de cada uma das técnicas são expressos em uma linguagem algorítmica genérica, com propriedades e semântica similares às de um subconjunto de uma linguagem procedural imperativa como C ou Pascal. Como notação adicional para estes algoritmos, os campos de um objeto composto são referenciados, no texto do algoritmo, na forma *Obj.campo*: o nome da variável que é a referência ao objeto *Obj*, seguido por um ponto e o nome do campo em questão. Por exemplo, se um objeto referenciado pela variável R possui campos chamados *nome* e $telefone$, a notação $R.nome$ indica uma referência para o campo *nome* do objeto referenciado por R. Nos algoritmos descritos, o campo com nome *children* de um objeto O representa o conjunto de objetos apontados por O, e o campo *size* representa o tamanho do objeto, em *bytes*.

Para mais detalhes sobre as três técnicas descritas abaixo e os algoritmos utilizados o leitor deve consultar o livro de Jones e Lins [21].

2.3.1 Marcar-e-varrer

O primeiro algoritmo para reciclagem automática de memória foi uma técnica de *rastreamento*: o método Marcar-e-Varrer (*mark-sweep* ou *mark-scan*), desenvolvido por J. McCarthy em 1960 para a linguagem Lisp [35]. Esta é uma técnica de rastreamento pois os objetos que são acessíveis a partir do conjunto raiz são explicitamente rastreados, partindo-se dos objetos que são raízes e seguindo-se as referências em ponteiros. Desta forma, o método de marcar-e-varrer é baseado em separar, através de um processo de varredura ou rastreamento, os objetos no *heap* nas duas classes de objetos ativos e inativos, e posteriormente separar os inativos para reaproveitamento. O momento de realizar essa varredura é normalmente quando o programa requer a alocação de um novo objeto e o sistema de gerenciamento de memória determina que não há mais memória disponível, o que dispara um *ciclo de coleta*, durante o qual fica suspenso o processo do usuário. O algoritmo de alocação está na Figura 2.1.

O algoritmo Um ciclo de coleta é composto por duas fases: marcação e varredura, como mostrado no algoritmo da Figura 2.2. Na marcação, as células acessíveis são *marcadas* como em uso; na varredura, as células que não estão marcadas são recolhidas. Para guardar o estado de marcação, cada célula possui um campo – que pode ser apenas um bit – reservado,

o *campo de marcação*, que recebe o nome de `mark`. O fim da fase da marcação é identificado quando não existem mais células acessíveis que não tenham sido marcadas (veja a Figura 2.3). Depois que a marcação termina, todo o *heap* é varrido linearmente e as células que não estão marcadas são identificadas como livres e podem ser reaproveitadas, ou seja, incluídas na lista livre. O algoritmo está detalhado na Figura 2.4.

```
New() =
   if free_list is empty
      mark_sweep()
   newcell = allocate()
   return newcell
```

Figura 2.1: *Alocação no algoritmo de marcar-e-varrer*

```
mark_sweep() =
   for R in Roots
      mark(R)
   sweep()
   if free_list is empty
      abort "Nao foi possivel alocar memoria"
```

Figura 2.2: *O algoritmo Marcar-e-Varrer*

```
mark(N) =
   if N.mark == unmarked
      N.mark = marked
      for M in M.children
         mark(*M)
```

Figura 2.3: *O algoritmo de marcação*

2.3.2 Coleta por cópia

A coleta por cópia é outra técnica de rastreamento, tendo portanto similaridades com a coleta marcar-e-varrer. O funcionamento do algoritmo, entretanto, é diferente: a idéia é que o *heap* é dividido em duas partes, chamadas de *semi-espaços*; em um dado momento, apenas um deles é utilizado. Quando não há mais memória disponível no semi-espaço em uso, o coletor identifica todos os objetos ativos, copiando-os para o outro semi-espaço. Depois disso, o papel

```
sweep() =
   N = Heap_bottom
   while N < Heap_top
      if N.mark == unmarked
         free(N)
      else
         N.mark = unmarked
      N = N + N.size
```

Figura 2.4: *O algoritmo de varredura*

dos dois semi-espaços é trocado até o próximo ciclo de coleta. Tradicionalmente os dois semi-espaços recebem os nomes de espaço de origem (*from space*) e espaço de destino (*to space*). O espaço de destino é sempre o semi-espaço que está em uso, enquanto que o espaço de origem não é utilizado e contém sempre os objetos anteriores ao momento da última coleta. Na figura 2.5 é mostrado o algoritmo de inicialização para o coletor de cópia; `Tospace` e `Fromspace` são os espaços de destino e origem, respectivamente; `free` é um ponteiro para o primeiro endereço livre para alocação.

```
init() =
    Tospace = Heap_bottom
    space_size = Heap_size / 2
    top_of_space = Tospace + space_size
    Fromspace = top_of_space + 1
    free = Tospace

New(n) =
    if free + n > top_of_space
        flip()
    if free + n > top_of_space
        abort "Nao foi possivel alocar memoria"
    newcell = free
    free = free + n
    return newcell
```

Figura 2.5: *Inicialização e alocação na coleção por cópia*

O algoritmo Inicialmente, os papéis dos dois espaços são trocados pela rotina `flip`. Cada célula acessível a partir das raízes é copiada do `Fromspace` para o `Tospace` pela rotina `copy(R)`.

É necessário ter um cuidado com as referências que restam durante o processo de cópia para preservar a topologia do grafo da memória, pois caso contrário algumas células poderiam ser copiadas mais de uma vez. A forma usual de evitar esse problema é através de *endereços de encaminhamento*. Em cada objeto que é copiado deixa-se um endereço de encaminhamento no `Fromspace` que aponta para o novo endereço deste no `Tospace`. Sempre que uma célula no `Fromspace` for visitada durante a cópia, o algoritmo verifica se ela já foi copiada, e neste caso utiliza o endereço de encaminhamento armazenado; caso contrário, a célula é copiada para o `Tospace` e o endereço de encaminhamento é armazenado. Note-se que o endereço é armazenado *antes* da cópia efetiva – isso assegura a terminação do algoritmo e a preservação da topologia do grafo. O endereço de encaminhamento é guardado na própria célula; no algoritmo, assume-se que ele é armazenado em um campo chamdo `forward`.

```
flip() =
    Fromspace, Tospace = Tospace, Fromspace
    top_of_space = Tospace + space_size
    free = Tospace
    for R in Roots
        R = copy(R)
```

Figura 2.6: *A rotina de troca de espaços*

```
copy(P) =
    if atomic(P) or P == nil          -- se P é atomico, ele é uma folha do grafo
        return P
    if not forwarded(P)
        n = P.size
        P' = free                     -- obtém memória no espaço de destino
        free = free + n
        P.forward = P'
        P' = copy(P)                  -- copia o objeto para o espaço de destino
    return P.forward
```

Figura 2.7: *Algoritmo de coleta por cópia*

2.3.3 Contagem de referências

A contagem de referências é um método direto, no qual um objeto é identificado como inativo no momento em que a última referência para ele deixa de existir. A idéia do algoritmo

é simples: ter em cada objeto um campo adicional para manter o número de objetos que o referenciam, ou seja, um contador de referências. Quando este contador chega ao valor zero em um objeto O, isso significa que não há mais referências para O, e a memória ocupada por ele pode ser reaproveitada imediatamente. A maior virtude deste algoritmo é a simplicidade para determinar se uma célula está ativa ou não. Também é uma técnica naturalmente incremental, distribuindo a sobrecarga do gerenciamento de memória durante toda a execução do programa.

A invariante deste algoritmo é que o contador de referências de uma célula C tenha valor igual ao número de outras células ativas que apontam para C; o gerenciador de memória deve garantir a manutenção dessa invariante. Para isso, é necessário alterar o valor do contador em cada operação com ponteiros. Assim como no algoritmo marcar-e-varrer, as células livres que podem ser alocadas são mantidas em uma estrutura de lista encadeada chamada de *lista livre*.

O algoritmo Células livres têm um contador de referências com valor zero. Quando uma nova célula é alocada, seu contador de referências é atualizado para 1, e cada vez que uma referência é criada ou alterada para apontar para esta célula, seu contador é incrementado. Da mesma forma, cada referência destruída ou alterada para não mais apontar para a célula causa uma redução de 1 no valor do contador. Se o contador chegou a zero, a célula se encontra inacessível e pode ser retornada à lista livre.

A Figura 2.8 mostra o algoritmo da alocação de células na contagem de referências. A função de baixo nível `allocate` obtem uma célula livre da lista, enquanto que `New` usa `allocate` e inicializa a célula obtida; o contador de referências é acessado através do campo `rc`. A rotina `Create` é utilizada quando uma nova referência é criada para o objeto `T`. Os algoritmos para remoção e alteração de ponteiros são mostrados na Figura 2.9; `Update` altera o local apontado por `R` para apontar para `S`, ajustando as referências de acordo, enquanto `Delete` é usada quando uma referência para o objeto `T` é destruída; a rotina `free` é chamada por `Delete` para mover uma célula inativa para a lista livre.

O programa que utiliza o sistema de contagem de referências deve utilizar as rotinas `New`, `Create`, `Delete` e `Update` sempre que for necessário operar com ponteiros, para garantir que o invariante da contagem de referências seja mantido.

```
allocate() =
    newcell = free_list
    free_list = free_list.next
    return newcell

New() =
    if free_list == nil
        abort "Não foi possível alocar memória"
    newcell = allocate()
    newcell.rc = 1
    return newcell

Create(T) =
    T.rc = T.rc + 1
```

Figura 2.8: *Alocação de células e criação de ponteiros na contagem de referências*

2.3.4 Comparações entre as técnicas

Após apresentar o funcionamento das três técnicas principais para o gerenciamento automático da memória, comparamos agora as suas características, destacando as vantagens e desvantagens relativas de cada uma.

2.3.4.1 Marcar-e-varrer

As técnicas que utilizam varredura – marcar-e-varrer e coleta por cópia – têm duas vantagens principais sobre a contagem de referências: a primeira é que ciclos são recuperados naturalmente, sem necessidade de medidas especiais; a segunda é que nenhuma sobrecarga é imposta às operações com ponteiros. Por outro lado, a coleta nestas técnicas impõe uma parada completa do programa principal (do usuário) enquanto o coletor trabalha – na coleta marcar-e-varrer, esse trabalho consiste na marcação das células e na varredura completa do *heap*. Com o aumento do tamanho das memórias, verificou-se que programas que utilizavam um coletor marcar-e-varrer gastavam uma porcentagem significativa do tempo de execução no coletor [21]. Além disso, a pausa causada pela varredura pode ser indesejável ou inaceitável em certos sistemas, como os de tempo-real.

Entretanto, se o tempo de resposta não é crucial, esta técnica pode oferecer um desempenho superior ao da contagem de referências. Mesmo assim, o custo da coleta é alto: cada célula ativa é visitada durante a fase de marcação, e todas as células do *heap* são visitadas

```
free(N) =
    N.next = free_list
    free_list = N

Delete(T) =
    T.rc = T.rc - 1
    if T.rc == 0
        for U in T.children
    delete(*U)
free(T)

Update(R, S) =
    delete(*R)
    S.rc = S.rc + 1
    *R = S
```

Figura 2.9: *Remoção e alteração de ponteiros na contagem de referências*

durante a varredura. Portanto, a complexidade assintótica do algoritmo é proporcional ao tamanho do *heap*. Outros problemas de desempenho são a fragmentação da memória e a perda da localidade de referências associada a isso; o procedimento de varredura prejudica o desempenho em sistemas com hierarquias de memória, causando *thrashing*. Ainda outro problema é que o desempenho de um coletor marcar-e-varrer depende da porcentagem de ocupação do *heap*, pois quanto mais ocupado mais freqüentes serão as coletas; já a contagem de referências não sofre degradação pelo aumento da taxa de ocupação.

Em relação à coleta por cópia, a técnica de marcar-e-varrer tem a vantagem principal de utilizar todo o *heap*, enquanto que a coleta por cópia divide-o em dois semi-espaços, como será visto na próxima seção; além disso, o algoritmo de marcar-e-varrer não sofre de cinetose (*motion sickness*), uma vez que as células em uso não são deslocadas durante a coleta. Por outro lado, coletores marcar-e-varrer causam uma maior fragmentação do *heap* em relação aos coletores por cópia, já que estes últimos compactam a memória. Fragmentação promove uma piora na localidade de referência na memória ocupada, degradando o desempenho. Além disso, a coleta por cópia não precisa varrer a memória toda.

Resumindo, as principais vantagens da técnica de marcar-e-varrer são:

▷ utiliza toda a memória;

▷ precisa de apenas um bit por nó;

- não impõe nenhum custo nas operações de ponteiros;
- não sofre de *motion sickness*.

Já suas desvantagens mais acentuadas são:

- tempo de pausa para varredura e coleta;
- varre todo o espaço de endereçamento do processo, perdendo a localidade de referências;
- fragmenta o espaço de memória.

2.3.4.2 Coleta por cópia

As vantagens da coleta por cópia sobre a contagem de referências e a coleta marcar-e-varrer a tornaram amplamente usada. Custos de alocação são extremamente baixos; a fragmentação é eliminada pela compactação da estrutura de dados ativos. O custo da alocação nas outras técnicas é muito maior, especialmente se objetos de tamanhos variáveis forem utilizados.

O custo mais imediato da coleta por cópia é o uso de dois semi-espaços: o espaço de endereçamento necessário é o dobro em comparação aos coletores que não utilizam cópia. Pode-se argumentar que o uso de memória virtual elimina o problema; mesmo assim, o dobro do espaço de memória deve ser alocado e reservado para o programa. Ao mesmo tempo, esses custos devem ser pesados contra as vantagens da compactação. Em sistemas atuais, onde a diferença de desempenho entre as CPUs e as memórias é crescente, ter uma boa localidade de referência nos dados utilizados pelo programa é cada vez mais importante. Por outro lado, o algoritmo de coleta por cópia sofre de *motion sickness*, uma vez que as células copiadas mudam de endereço; tal característica pode causar dificuldades em alguns tipos de aplicação.

Resumindo, temos como vantagens desta técnica:

- evita fragmentação por copiar os objetos;
- tempo de coleta menor que o de marcar-e-varrer;
- não necessita de espaço extra para cada célula, pois o endereço de encaminhamento pode ser armazenado em algum outro campo.

Já como desvantagens, podemos citar:

- perde muito tempo copiando objetos;
- pode causar muitas falhas no sistema de memória virtual durante o processo de cópia entre semi-espaços;

▷ sofre de *motion sickness*.

Com relação à complexidade temporal do processo de coleta, ela é proporcional ao grafo de objetos na memória, e não ao tamanho do *heap*, como no caso do algoritmo marcar-e-varrer.

2.3.4.3 Contagem de referências

A principal vantagem da contagem de referências é distribuir a carga do gerenciamento de memória ao longo da execução do programa, sem a necessidade de parar toda a computação útil para isso. Em comparação, as outras técnicas indiretas como marcar-e-varrer paralisam o programa do usuário enquanto buscam células inativas que podem ser coletadas. Isso pode ser uma vantagem em sistemas que precisem manter um tempo de resposta garantido, como sistemas de tempo-real. Entretanto, a carga de gerenciamento é distribuída de maneira não uniforme – a remoção de uma célula pode ter custo proporcional ao tamanho do sub-grafo sob a célula, por exemplo. De fato, por este motivo as pausas em um sistema de contagem de referências podem se tornar comparáveis às das outras técnicas [9].

Estudos empíricos em várias linguagens apontam que a maioria das células não são compartilhadas e ficam ativas por pouco tempo [21, 24]. O algoritmo de contagem de referências permite o reaproveitamento imediato dessas células, minimizando a ocupação do programa do usuário e ajudando a evitar que a memória seja paginada no disco[2]. Além disso, esse conhecimento de células que podem ser reaproveitadas imediatamente pode ser utilizado em otimizações, quando uma nova célula é similar a uma que está sendo liberada. Por fim, a liberação imediata de células inativas é útil para garantir que ações de *finalização* em objetos sejam determinísticas [8].

Mesmo com essas vantagens, o algoritmo possui uma série de problemas que inibiram seu uso. A desvantagem mais séria é o alto custo sofrido nas atualizações de ponteiros. Nas técnicas indiretas as operações com ponteiros não possuem custos adicionais. Outro impacto sobre o desempenho temporal do programa ocorre por causa da liberação imediata de células inativas e a recursividade na liberação das células – ver a rotina `Delete` na Figura 2.9. Suponha que uma célula C seja a raiz de uma estrutura de dados com grande número de células, ou seja, todos os caminhos do conjunto raiz para qualquer célula da estrutura passam por C. No momento que C se torna inativa, toda a estrutura também estará inativa e pode ser liberada;

[2] Em geral deve-se utilizar registradores ou alocação em pilha para objetos desta natureza, pois a geração de uma célula que estará ativa durante um curto espaço de tempo tem custo computacional muito maior que a alocação de valores nestes locais.

entretanto, isso torna o tempo ocupado com a liberação maior do que seria para liberar apenas uma célula. De fato, como o sub-grafo sob uma célula qualquer é potencialmente ilimitado, não se pode prever qual o tempo gasto durante a liberação recursiva, impedindo o uso desta versão do algoritmo para sistemas de tempo real, por exemplo. Medições realizadas em sistemas reais demonstram que as pausas causadas pela liberação recursiva podem ser comparáveis às pausas em coletores de rastreamento [9], mas há técnicas para contornar tal problema [47].

Outra desvantagem é o alto grau de acoplamento entre o coletor e o resto do programa. O programa do usuário (ou compilador da linguagem) deve levar em conta a existência dos contadores de referências, e deve usar operações indiretas com ponteiros para que a invariante do algoritmo seja mantida. As técnicas indiretas podem ser implementadas de maneira menos acoplada ao programa do usuário.

Há também um custo espacial: cada célula precisa guardar um contador além dos dados do programa. Dependendo do tamanho das células, essa sobrecarga pode ser mais ou menos significativa. Entretanto, existem técnicas para minimizar o tamanho do contador, até mesmo usando apenas um bit.

Estruturas de dados cíclicas Entretanto, o maior problema da contagem de referências é a detecção de células inativas em grafos cíclicos. Ciclos de células que se referenciam umas às outras mas que não possuem nenhuma referência que parta das raízes são inativas, e portanto poderiam ser reaproveitada. Mas no algoritmo clássico de contagem de referências, estas células sempre terão registrado um número de referências pelo menos igual a 1.

Existem algumas possibilidades para resolver este problema, como será visto no próximo capítulo. Em geral, combina-se a contagem de referências com alguma técnica de rastreamento, como marcar-e-varrer, para detectar os ciclos.

capítulo 3

Contagem de Referências Cíclicas

Como visto no capítulo anterior, a técnica original de contagem de referências não trata corretamente as estruturas cíclicas, pois qualquer conjunto de objetos que se referencie em um ciclo de ponteiros não é recuperado, mesmo que seja lixo. Este problema foi percebido por McBeth em 1963 [34]. A solução mais comum, durante os anos seguinte, foi utilizar uma técnica híbrida, combinando a contagem de referências com alguma técnica baseada em rastreamento – normalmente marcar-e-varrer – para detectar e recuperar ciclos. Mas isso requer combinar dois sistemas globais para gerenciamento de memória, o que aumenta a complexidade geral e introduz redundância, já que os dois sistemas são relativamente independentes.

Neste capítulo é apresentada uma solução eficiente para o problema, baseada em um algoritmo que foi sucessivamente melhorado a partir da idéia inicial, tratada na seção 3.1. Em seguida, será tratada a versão mais recente e mais eficiente do algoritmo até agora, na seção 3.2. Por fim, o capítulo conclui com um exemplo para ilustrar melhor o funcionamento do algoritmo.

3.1 Detecção de ciclos usando uma varredura local

A primeira solução geral para o problema dos ciclos no algoritmo da contagem de referências foi publicada em 1990 no artigo de Martinez, Wachenchauzer e Lins [32]. A idéia do algoritmo neste artigo é realizar uma varredura local nos nós do grafo da memória que podem ser ciclos, detectando corretamente os ciclos que não estão transitivamente conectados

ao conjunto raiz. Um objeto O é considerado candidato para a varredura local quando seu contador de referências é decrementado para um valor maior que 0. Neste caso, o nó era compartilhado e existem duas possibilidades para a referência remanescente: ou ela vem de algum objeto ligado transitivamente às raízes, e O ainda está ativo; ou vem de um objeto ligado ciclicamente a O, e neste caso todos os objetos ligados ao ciclo estão inativos se nenhum deles estiver conectado a alguma raiz. O algoritmo então procede fazendo uma varredura no sub-grafo iniciado em O, removendo temporariamente as referências que podem ter origem em um ciclo; ao fim deste processo, se não restar nenhuma referência nos nós desse sub-grafo, pode-se concluir que os nós fazem parte de um ciclo, e podem ser recuperados.

Posteriormente, Lins publicou uma série de artigos [25, 28, 30] detalhando melhorias no algoritmo original. A principal melhoria é adiar a varredura local realizada em nós compartilhados, pois verifica-se empiricamente que a maioria dos nós compartilhados não faz parte de um ciclo (o artigo de Levanoni e Petrank [24] contém evidências recentes). Neste caso, o algoritmo original estaria desperdiçando o trabalho de varredura local, na maior parte do tempo. Já quando a varredura é adiada, muitas vezes é possível determinar sem esforço adicional se o nó não faz parte de um ciclo.

3.2 Contagem eficiente de referências cíclicas

Segue a descrição da versão mais recente do algoritmo para contagem de referências cíclicas [26]. O leitor interessado na evolução do algoritmo pode consultar os artigos [25, 28, 30] ou a dissertação de mestrado de Salzano Filho [15].

Além do contador de referências, é preciso armazenar em cada objeto um campo indicando sua cor. Três cores são possíveis: verde indica que a célula está em uso; vermelho é usado para células que estão sendo varridas localmente pelos algoritmos de verificação de ciclos; e células com a cor preta estão marcadas para uma varredura local posterior. Uma estrutura de dados, chamada de *status analyser*, é utilizada para guardar ponteiros para todas as células que poderão ser varridas.

Novas células são obtidas através da função `New`, mostrada na figura 3.1. A nova célula é obtida da lista livre, se esta não estiver vazia; caso contrário, o algoritmo dispara uma varredura do *status analyser* para tentar recuperar células fora de uso. Em todo caso, se uma célula é obtida, seu contador de referências inicia com o valor 1 e sua cor como verde. A função `allocate`, que retira uma célula da lista livre, é idêntica à da Figura 2.8. Quando é

```
New() =
   if free-list not empty then
      newcell = allocate()
   else
      if status-analyser not empty then
         scan_status_analyser ()
         newcell = New()
      else
         abort "Não há células disponíveis"
   newcell.rc = 1
   newcell.color = green
   return newcell
```

Figura 3.1: *Operação para obter uma nova célula na contagem de referências cíclicas*

```
Create(T) =
   T.rc = T.rc + 1
   T.color = green
```

Figura 3.2: *Criação de uma referência na contagem de referências cíclicas*

necessário criar uma nova referência para a célula `T`, chama-se a rotina `Create`, que apenas incrementa o contador de referências de `T` e muda sua cor para verde; isso é feito pois se `T` só pode receber uma nova referência se estiver ativa. Note que isto pode ocorrer enquanto `T` está no *status analyser*, o que significa que quando esta estrutura for varrida, a condição de `T` já será conhecida e não será necessário fazer uma varredura local no seu sub-grafo.

A outra parte da interface do sistema de contagem de referências com o programa são as rotinas para alteração e remoção de referências. A rotina `Update` para a contagem de referências cíclicas é a mesma para a contagem de referências tradicional, mostrada na figura 2.9. Entretanto, `Delete` é diferente: caso o contador da célula tenha valor 1 na chamada a `Delete`, significa que ela será liberada, e seu sub-grafo precisa ser ajustado, da mesma forma que no algoritmo tradicional; entretanto, se o contador tinha valor maior que 1, a célula era compartilhada e deve ser examinada posteriormente para verificar a existência de ciclos. Por isso, a célula neste caso é pintada com a cor preta e adicionada ao *status analyser*.

A parte da técnica que trata da detecção de ciclos está expressa em um conjunto de rotinas inter-relacionadas. Como visto na Figura 3.1, a rotina `New` pode disparar uma varredura do *status analyser* caso não haja mais células disponíveis na lista livre. Esta varredura é feita

```
Delete(S) =
   if (S.rc == 1) then
      for T in S.children do
         Delete(T)
      S.color = green
      Link_to_free_list(S)
   else
      S.rc = S.rc - 1
      if S.color != black then
         S.color = black
         add_to_status_analyser(S)
```

Figura 3.3: *Remoção de um ponteiro na contagem de referências cíclicas*

```
scan_status_analyser () =
   S = select_from(status_analyser)
   if S == nil then return
   if S.color == black then
      Mark_red(S)
   if S.color == red && S.rc > 0 then
      scan_green(S)
   scan_status_analyser ()
   if S.color == red then
      Collect(S)
```

Figura 3.4: *Rotina para varrer o* status analyser

pela rotina `scan_status_analyser`, mostrada na Figura 3.4. Esta rotina examina todas as células no *status analyser* em alguma ordem, disparando uma varredura local em todas as que ainda tiverem a cor preta. Esta varredura é feita pela rotina `Mark_red`; ao fim da varredura, se a célula tiver cor vermelha mas contador de referências maior que zero, isso significa que há alguma referência externa (transitivamente ligada às raízes) para ela, e ela está ativa. Neste caso, é chamada a rotina `scan_green`, para reestabelecer a cor verde na célula e em seu sub-grafo. A chamada recursiva em `scan_status_analyser` indica que o processo continua com outras células; ao final, as células que ainda estiverem com a cor vermelha podem ser consideradas inativas e devolvidas à lista livre, o que é feito pela rotina `Collect`.

`Mark_red` é a rotina responsável pela varredura local de células compartilhadas, a fim de detectar ciclos. Quando chamada para examinar uma célula `S`, primeiro a cor de `S` é alterada para vermelho. Em seguida, `Mark_red` reduz o contador de referências de todas as células

```
Mark_red(S) =
   if S.color != red then
      S.color = red
      for T in S.children do
         T.rc = T.rc - 1
      for T in S.children do
         if T.color != red then
            Mark_red(T)
         if T.rc > 0 && T not in status-analyser then
            add_to_status_analyser(T)
```

Figura 3.5: *Rotina para marcar células de vermelho*

apontadas por `S`; isso é feito para remover referências internas a um ciclo ou sub-grafo local. Em seguida, as células apontadas por `S` são varridas através da chamada recursiva a `Mark_red`, caso elas já não tenham cor vermelha, o que significa que já foram examinadas. Após a chamada recursiva, toda célula apontada por `S` que continua com um contador de referências maior que zero é adicionada ao *status analyser*, para que seja examinada posteriormente. Isto é feito para identificar outros pontos críticos no grafo da memória que possam indicar a presença ou ausência de ciclos, possivelmente acelerando o processo de varredura.

Na varredura local com `Mark_red` as células de um sub-grafo em análise têm seus contadores reduzidos para eliminar o papel das referências internas ao sub-grafo. Depois disso, se nenhuma célula no sub-grafo possuir um contador com valor maior que zero, pode-se concluir que o mesmo perfazia um ciclo e pode ser coletado; isso acontece na chamada a `Collect` dentro de `scan_status_analyser` (ver Figura 3.4). Entretanto, se alguma célula possui contador maior que zero, isso indica que há pelo menos uma referência externa ao sub-grafo dessa célula, e todas as células nesse sub-grafo podem ser consideradas ativas. Para reestabelecer a cor verde nessas células, e reconsiderar as referências internas ao sub-grafo, é chamada a rotina `Scan_green`, detalhada na Figura 3.6. Quando chamada em uma célula `S`, `scan_green` simplesmente muda a cor de `S` para verde e aumenta a contagem de referências de todas as células apontadas por `S`, chamando a si mesma recursivamente nessas. Isso é suficiente para desfazer o trabalho feito por `Mark_red` caso as células sejam ativas.

Finalmente, `Collect` é responsável pela coleta das células que foram determinadas como inativas após a varredura com `Mark_red`. Neste caso, o processo de coleta para uma célula `S` consiste em remover a referência de `S` para todas as células apontadas por ela, e devolver `S` à

```
Scan_green(S) =
   S.color := green
   for T in S.children do
      T.rc = T.rc + 1
      if T.color != green then
         scan_green(T)
```

Figura 3.6: *Rotina para varrer células pintadas de verde*

```
Collect(S) =
   for T in S.children do
      Delete(T)
      if T.color = red then
         Collect(T)
   S.rc = 1
   S.color = green
   add_to_free_list(S)
```

Figura 3.7: *Rotina para coleta das células inativas*

lista livre. Se alguma célula apontada por `S` também tiver cor vermelha, ela pode ser coletada similarmente; isso é indicado pela chamada recursiva a `Collect`. Desta forma, um sub-grafo considerado inativo, ou seja, um ciclo sem referências externas, pode ser recuperado.

3.3 Exemplos

Nesta seção são apresentados dois exemplos simples do algoritmo de contagem de referências cíclicas em funcionamento, para ilustrar a técnica. O primeiro mostra um ciclo que contém uma referência externa, e portanto ainda está ativo, enquanto o segundo mostra uma situação de um ciclo que é lixo e pode ser recuperado.

3.3.1 Exemplo de um ciclo ativo

A Figura 3.8 mostra a situação inicial para o primeiro exemplo. Dois objetos, X e Y, apontam para um ciclo de três outros objetos. Considera-se que X e Y estão transitivamente conectados ao conjunto raiz, embora isso não seja mostrado na figura. Os objetos do ciclo são mostrados com o valor de seus contadores de referência; o valor dos contadores de X e Y não é importante para o exemplo. Todos os objetos estão com cor verde, pois todo conjunto está ativo.

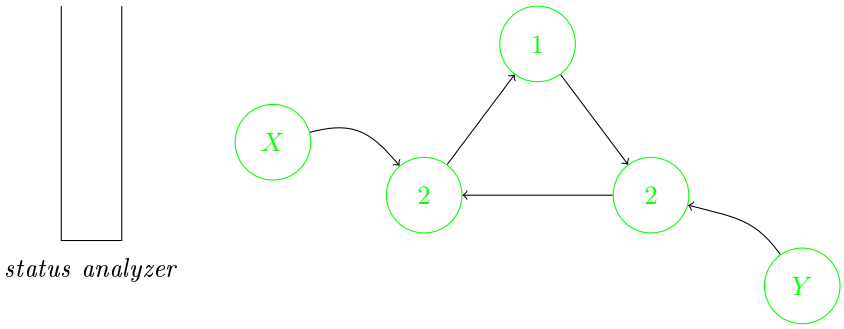


Figura 3.8: *Exemplo 1: situação inicial*

Em seguida, a referência de X para o ciclo é retirada, o que muda o contador do objeto mais à esquerda no ciclo de 2 para 1; neste momento o objeto é alterado para a cor preta e adicionado ao *status analyser*, para análise posterior. Esta situação é mostrada na Figura 3.9. Supondo que a situação local deste sub-grafo da memória não mude até uma varredura do *status analyser*, o nó de cor preta na figura será eventualmente escolhido em `scan_status_analyser`, que verificará sua cor como preta e chamará `Mark_red`. Esta rotina irá examinar o sub-grafo começando no nó à esquerda, pintando-o de vermelho, decrementando o contador de referências nos nós descendentes, e chamando `Mark_red` recursivamente. A Figura 3.10 mostra a situação após a rotina `Mark_red` ter concluído nesta parte do subgrafo. É importante notar também que, nessa figura, o nó do ciclo que fica com o contador de referências igual a 1 é adicionado ao *status analyser* por `Mark_red`.

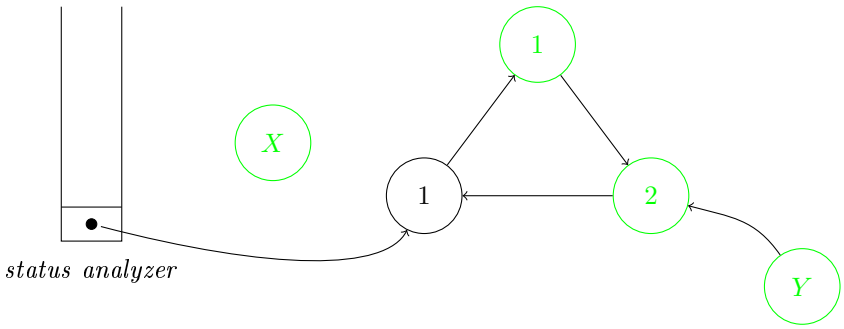


Figura 3.9: *Exemplo 1 após remoção da referência*

Tendo `Mark_red` terminado o trabalho de análise local no sub-grafo, o controle retorna para `scan_status_analyser`, que continua a analisar células até eventualmente chegar na célula do ciclo que tem o contador de referências igual a 1. Neste momento, como a cor da célula é vermelha e seu contador é maior que zero, é chamada a rotina `scan_green` para

desfazer o que foi feito durante a varredura em `Mark_red`, começando pelo nó com contador igual a 1. Então, `scan_green` muda novamente a cor das células vermelhas para verde e reestabelece a contagem correta de referências. A situação final, após o fim de `scan_green`, é mostrada na Figura 3.11

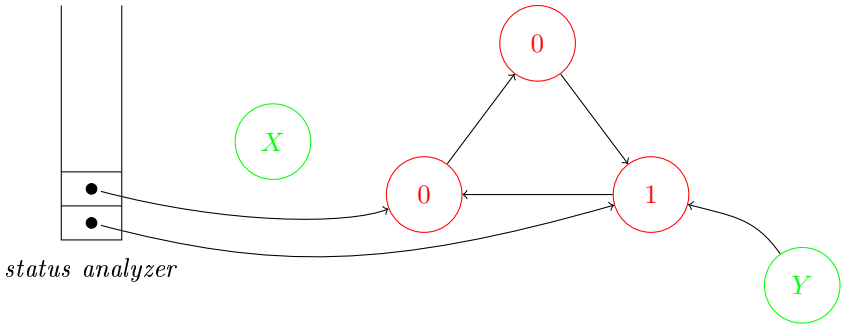


Figura 3.10: *Exemplo 1 após a varredura local e marcação em vermelho*

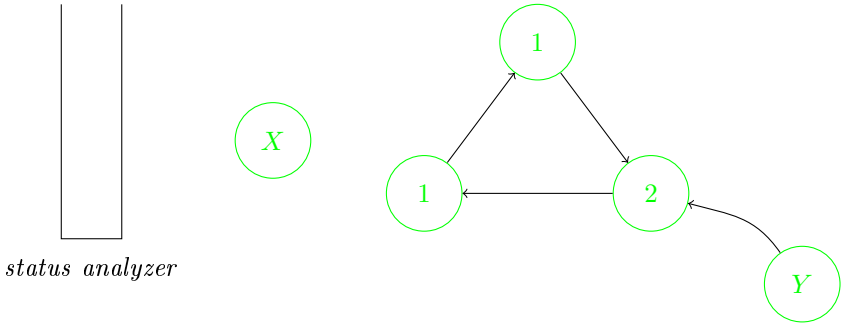


Figura 3.11: *Exemplo 1 após a recuperação das células para verde*

3.3.2 Exemplo de um ciclo inativo

O próximo exemplo mostra uma situação similar ao anterior, mas desta vez o ciclo não possui referências externas, e portanto pode ser liberado. A Figura 3.12 mostra a situação inicial.

A referência de X para o ciclo é retirada, o que causa mudança no contador de referências na célula afetada, do valor 2 para 1. A mesma célula é adicionada ao *status analyser*. Esta é a situação mostrada na Figura 3.13.

Novamente, supondo que a situação nesta região do grafo da memória não se altere até uma chamada a `scan_status_analyser`, esta rotina eventualmente tomará a célula de cor preta na Figura 3.13 para análise, chamando `Mark_red` para varrer seu sub-grafo. As células conectadas

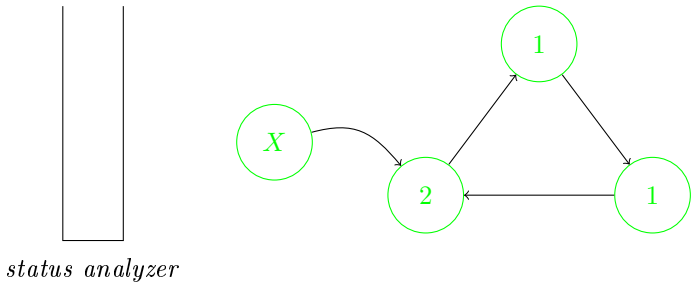


Figura 3.12: *Exemplo 2: situação inicial*

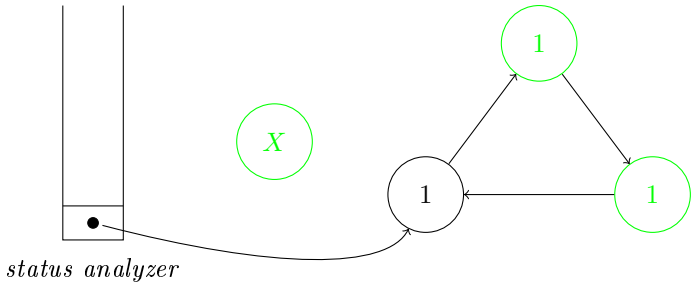


Figura 3.13: *Exemplo 2 após retirada da referência*

ao ciclo são então pintadas de vermelho e seus contadores de referência decrementados em uma unidade, para desconsiderar as referências internas. A situação após o final da varredura local é mostrada na Figura 3.14. Note-se que nesse caso não resta nenhuma célula no ciclo com contador de referências maior que zero, o que indica, corretamente, que o ciclo está inacessível a partir das raízes, e portanto é lixo.

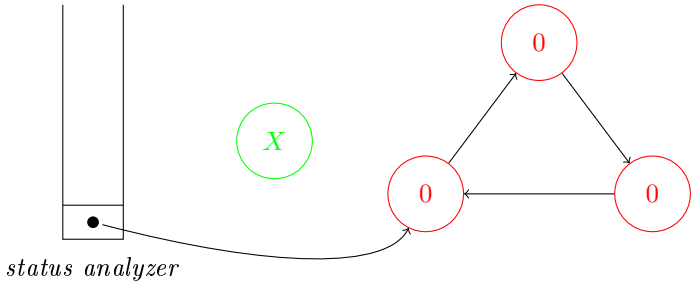


Figura 3.14: *Exemplo 2 após varredura local*

Após `Mark_red`, o controle volta para `scan_status_analyser`, que continuará analisando células. Como o ciclo foi desconectado do conjunto raiz e nenhuma de suas células está no *status analyser*, sua situação não muda com relação à Figura 3.14. Depois de terminar com

todas as células que precisavam de análise, a rotina `scan_status_analyser` volta a considerar as células já analisadas para coletar as que permanecerem com a cor vermelha. Neste caso, a célula mais à esquerda no ciclo das figuras estava no *status analyser* e, como continua vermelha, será coletada pela chamada a `Collect`. Esta última rotina se encarregará de coletar todas as células no sub-grafo representado pelo ciclo, devolvendo-as à lista livre.

3.4 Combinando contagem de referências cíclicas com liberação procrastinada

Foi mencionado no Capítulo 2 que a liberação recursiva de células no algoritmo da contagem de referências pode causar pausas na atividade do programa, e que não é possível obter um limite superior para a duração destas. Uma solução comum para o problema é usar uma técnica conhecida como *liberação procrastinada* (*lazy deletion*), em que células inativas não são liberadas recursivamente, mas apenas adicionadas a uma estrutura de dados. Seja C uma célula que é raiz de um sub-grafo na memória, e C se torna inativa; ela é então adicionada a uma pilha de células livres, e seu sub-grafo não é examinado. Quando for necessário obter uma nova célula, C é liberada para este fim, e as células filhas de C no sub-grafo são adicionadas à pilha de células livres. Assim, o coletor realiza um número limitado de operações de cada vez, evitando longas pausas. Esta solução foi proposta primeiramente por Weizenbaum [47], para corrigir o que ele viu como uma assimetria no algoritmo de contagem de referências original – a liberação é a única operação sem um limite superior de tempo dispendido.

Esta solução pode ser integrada de maneira elegante com o algoritmo de contagem de referências cíclicas visto neste capítulo. Para este fim, o *status analyser* pode ser utilizado de forma similar à pilha de células livres. Basta que, ao encontrar uma célula com contagem zero, ela seja adicionada ao *status analyser* ao invés de liberada imediatamente para a lista livre. Então, ao varrer o *status analyser*, estas células podem ser liberadas incrementalmente, como na proposta de Weizenbaum. Estas células cuja liberação foi adiada têm contador de referências igual a zero e cor preta, para diferencia-las das que precisam de uma varredura local.

As alterações no algoritmo deste capítulo ficam restritas apenas às rotinas `Delete` e `scan_status_analyser`, que são mostradas na Figura 3.15. Na rotina `Delete`, ao invés da liberação recursiva como na Figura 3.3, a célula inativa é pintada de preto e adicionada ao *status analyser* com contador de referências igual a zero. Em `scan_status_analyser` é

```
scan_status_analyser () =
   S := select_from(status_analyser)
   if S.color == black then
      if S.rc > 0 then                -- celula para varredura local
         Mark_red(S)
      else                            -- celula de liberacao adiada
         for T in S.children do Delete(T)
         Link_to_free_list(S)
   if S.color == red && S.rc > 0 then
      scan_green(S)
   scan_status_analyser ()
   if S.color == red then
      Collect(S)

Delete(S) =
   if (S.rc == 1) then
      S.rc = 0
      S.color = black
      add_to_status_analyser(S)
   else
      S.rc = S.rc - 1
      if S.color != black then
         S.color = black
         add_to_status_analyser(S)
```

Figura 3.15: *Alterações no algoritmo para usar liberação procrastinada*

que uma célula posteriormente inativa será liberada, adicionando-a à lista livre; note-se que `scan_status_analyser` então chama `Delete` nas células descendentes de uma célula liberada, mas `Delete` por sua vez irá adicionar as células que tenham se tornado inativas ao *status analyser*. Não há liberação recursiva, o que torna o tempo gasto com pausas limitado. Embora essa seja uma alteração simples no algoritmo, a sugestão de integrar o algoritmo de contagem de referências cíclicas de Lins [26] com a liberação não-recursiva de Weizenbaum [47] parece ser inédita, e o resultado demonstra afinidades entre os dois algoritmos. Até onde vai o conhecimento do autor desta dissertação, o algoritmo acima é o primeiro na literatura a detalhar esta integração.

Neste capítulo foi considerado o algoritmo para gerenciamento de memória baseado na contagem de referências cíclicas, mas apenas para o caso seqüencial, para computadores uni-

processados. O Capítulo 4 se ocupa de técnicas para realizar o gerenciamento de memória em um ambiente multiprocessado. O algoritmo seqüencial visto neste capítulo serve como base para as versões concorrentes da contagem de referências cíclicas.

capítulo 4

Coleta de Lixo em Sistemas Multiprocessados

O Capítulo 2 apresentou e comparou as três estratégias fundamentais para gerenciamento automático da memória, motivando o estudo em uma delas, a contagem de referências. Feito isto, o Capítulo 3 mostrou como podem ser solucionados dois problemas na técnica de contagem de referências: a impossibilidade de recuperar estruturas cíclicas, e os tempos de pausa sem limite superior na liberação de objetos. Com este trabalho preparatório realizado, neste capítulo é mostrado como o algoritmo do Capítulo 3 pode ser extendido para trabalhar de forma concorrente, em sistemas multiprocessados. O algoritmo concorrente tem como base a arquitetura de Lins [31], incluindo algumas alterações e análises de aspectos que não foram considerados no artigo original. Estas alterações são motivadas e comparadas com trabalhos recentes na área, tendo objetivos alinhados com a pesquisa atual.

4.1 Coordenação entre programa e gerenciador da memória

Os capítulos anteriores não especificaram explicitamente como ocorre a coordenação entre o coletor de lixo e o resto do programa, a parte que realiza a computação que é a função do programa em si. Do ponto de vista do coletor, tudo que o resto do programa faz é alterar o grafo da memória, e por isso a parte útil da computação é normalmente chamada de *mutador*.

Foi mencionado que os algoritmos estudados nos Capítulos 2 e 3 são adeqüados apenas

para sistemas seqüenciais ou uniprocessados, nos quais coletor e mutador executam em um mesmo processador, e portanto não podem executar simultaneamente. Neste contexto, os algoritmos de coleta de lixo que fazem rastreamento são naturalmente *coletores de parada*, pois param toda a computação no mutador enquanto a coleta é realizada; já o algoritmo da contagem de referências realiza suas operações de maneira intercalada com operações do mutador, o que torna a contagem de referências naturalmente *incremental*. Também foram propostos algoritmos de rastreamento que trabalham incrementalmente, dos quais alguns são apresentados no livro de Jones e Lins [21].

A questão então se torna como aproveitar a capacidade dos computadores multiprocessados. Para isso, é necessário comentar sobre a alocação de processos ou *threads* nos processadores disponíveis. Os sistemas de gerenciamento automático da memória devem, pela sua própria função, executar no mesmo espaço de memória do mutador, e isso implica que o coletor e o mutador serão separados em *threads* diferentes. Também é possível ter vários mutadores executando concorrentemente, um em cada *thread*.

Mesmo em um computador uniprocessado atual, é comum ter coletores de rastreamento executando em uma *thread* separada do mutador – ou mutadores. Como só há um processador nesse caso, apenas uma destas *threads* executa em um dado momento. Além disso, um coletor de parada requer que todas as *threads* de mutadores sejam paradas enquanto durar o ciclo de coleta. Embora a parada possa ser indesejável com relação ao tempo de resposta do programa, ao menos a utilização dos recursos computacionais – o processador – não é prejudicada pelo coletor: ele próprio só utiliza o processador durante um ciclo de coleta. No resto do tempo, o processador está disponível para os mutadores.

Já no caso multiprocessado um coletor de parada pode desperdiçar os recursos computacionais disponíveis. Imaginando um sistema com 32 processadores, ou um processador com 32 núcleos, isso garante 32 *threads* executando simultaneamente a qualquer momento. Se o coletor precisa parar todas as *threads* para realizar um ciclo de coleta, apenas um processador será utilizado enquanto durar o ciclo, o que desperdiça todos os outros 31 processadores disponíveis. Esse é o caso com o coletor de lixo atualmente utilizado na máquina virtual Java da Sun [24].

Deste problema surgem as duas formas principais de explorar os recursos computacionais em coletores de lixo que executam em sistemas multiprocessados. A primeira é manter a parada de todas as *threads*: o coletor ainda pára todas as *threads* de mutadores, mas executa

paralelamente, possivelmente utilizando todos os processadores disponíveis; isto é chamado de *coleta paralela*. A segunda solução é fazer com que o coletor execute assincronamente com as *threads* de mutadores, sem que seja necessário parar todas elas para realizar a coleta, ou parando-os apenas por uma fração do período de tempo de um ciclo. Estes são chamados de *coletores concorrentes*.

A transição natural dos coletores incrementais para computadores multiprocessados é transforma-los em coletores concorrentes, pelo fato das operações de coleta já executarem intercaladamente com as operações de mutadores. Entretanto, o problema que pode surgir neste caso é como evitar que as *threads* de mutadores e coletores interfiram um com o outro – uma solução típica, usando sincronização por semáforos, pode degradar significantemente o desempenho do sistema resultante [12]. A contagem de referências, como um método naturalmente incremental, está sujeito a estes problemas. O desafio passa a ser uma implementação eficiente da técnica de contagem de referências em sistemas multiprocessados, que dependa o mínimo possível de sincronização bloqueante, ou mesmo que seja *lock-free* [11].

4.1.1 Coletores paralelos *versus* concorrentes

A decisão entre usar um coletor paralelo ou um coletor concorrente reflete um conflito tradicional na ciência da computação, que é escolher entre maximizar a *vazão* ou minimizar a *latência* [40]. A vazão representa a média de número de instruções executadas por unidade de tempo, enquanto que a latência mede o tempo total de execução das instruções.

Nos coletores paralelos, o processo de coleta em si tende a prejudicar menos a vazão do programa, em relação aos coletores concorrentes. Embora os mutadores fiquem paralisados durante o ciclo de coleta, os processadores disponíveis serão eficientemente utilizados se a carga do coletor estiver bem balanceada. Em contrapartida, a latência do programa será certamente aumentada, por causa das paradas que o coletor impõe aos mutadores.

Já nos coletores concorrentes a situação é invertida, como é o caso com a contagem de referências. Eliminando a possibilidade de pausas ilimitadas causadas pela liberação recursiva, os tempos de pausa (e portanto a latência) normalmente são bem pequenos com relação aos coletores paralelos de parada. Em alguns casos a latência em um coletor concorrente incremental pode chegar a ser duas ordens de magnitude menor que em um coletor paralelo de parada [24].

Mesmo que se utilize uma arquitetura de coletor que seja concorrente e paralela, um dos

dois aspectos deverá ser privilegiado: maximizar a vazão ou minimizar a latência. Em programas que interagem com o usuário ou têm limites de tempo de resposta – sistemas de tempo-real, por exemplo – o objetivo é reduzir a latência, enquanto que programas que realizam processamento sem interação com usuário ou limites de tempo de resposta – programas científicos, processamento de imagens em lote – o objetivo é maximizar a vazão. No geral, o melhor desempenho do coletor de lixo dependerá do programa no qual ele é utilizado, e é tarefa do programador ajustar os parâmetros para obter seus objetivos de desempenho.

4.2 Contagem de referências em sistemas multiprocessados

Como já foi mencionado, a técnica de contagem de referências é naturalmente incremental, e portanto pode ser extendida naturalmente para um coletor concorrente. O problema é, então, a sincronização imposta pelo coletor, que pode degradar o desempenho do sistema significativamente. Mas a forma que a sincronização é necessária, e como isso pode afetar o desempenho geral do programa, depende da arquitetura específica que se utiliza. Por esse motivo, as seções seguintes descrevem e analisam três arquiteturas possíveis:

1. um coletor concorrente composto por um mutador e um coletor;
2. outro coletor concorrente, mas incluindo vários mutadores para apenas um coletor;
3. por último, um coletor concorrente e paralelo, composto por vários mutadores e vários coletores.

Essas arquiteturas serão descritas a seguir. Antes, entretanto, serão analisadas as dificuldades principais acarretadas pela existência de interferência entre coletor e mutador, em coletores por contagem de referências.

4.2.1 Interferência entre coletor e mutador

Um coletor por contagem de referências trabalha fortemente acoplado com o mutador (ou mutadores), o que acarreta que a ocorrência de interferências entre coletor e mutador se torna freqüente quando executados concorrentemente. De maneira informal e para os propósitos deste trabalho, uma *thread interfere* com outra quando executa alguma instrução que invalida uma asserção de alguma instrução na outra *thread*. Uma definição mais formal pode ser encontrada nos livros de Andrews [2, 3].

Especificamente, existem duas dificuldades principais ou intrínsecas da contagem de referências em sistemas multiprocessados, e mais uma dificuldade que decorre de uma característica comum da sua implementação:

1. a manutenção dos contadores de referências;
2. as alterações nos ponteiros;
3. a manipulação da lista livre.

A primeira dificuldade ocorre porque toda operação com ponteiros deve alterar o contador de referências de uma ou duas células, e portanto podem acontecer condições de corrida quando duas ou mais *threads* tentam alterar o contador da mesma célula. Por exemplo, seja uma célula C com contador de referências igual a um, e duas *threads* que executam concorrentemente as seguintes operações:

▷ a *thread* 1 adiciona uma referência a C

▷ a *thread* 2 remove uma referência a C

Claramente, o contador de referências de C deve ter, ao final das duas operações, o valor 1. Mas pode ocorrer que, no escalonamento do sistema, a *thread* 2 execute suas operações antes que a *thread* 1; neste caso, o contador de C chegará ao valor zero, e a célula será liberada imediatamente. Quando a *thread* 1 tentar adicionar uma referência a C, esta estará na lista livre, e estará sujeita a ser alocada para outro fim, o que é uma situação indesejável e pode tornar o sistema instável. Esse é um exemplo de interferência entre as *threads*, pois uma delas invalida uma asserção da outra. Muitas outras condições de corrida podem ocorrer, inclusive que duas *threads* se intercalem durante a alteração do valor de um contador, deixando-o com um valor diferente do que ambas esperam.

A dificuldade com relação à alteração dos ponteiros ocorre pois os mutadores não realizam diretamente essas alterações; é preciso utilizar as operações da interface do coletor para isso, já que toda alteração deve contabilizar as mudanças nos contadores de referências das células envolvidas. Entretanto, enquanto o programa dos mutadores pode conter o conhecimento de que células são compartilhadas por quais *threads*, o programa do coletor é genérico e não deve ter conhecimento dos arranjos de compartilhamento entre os mutadores. Isso se traduz em um problema de interferência entre mutadores no qual o coletor não tem o conhecimento de quais podem interferir com quais outros, então deve assumir o pior: que qualquer *thread* mutadora pode interferir com qualquer outra. Assim, para garantir que os ponteiros serão

alterados corretamente, uma solução direta para garantir a não-interferência é usar um *mutex* (semáforo para exclusão mútua) compartilhado entre o coletor e todos os mutadores, efetivamente serializando todas as alterações em ponteiros. Esta é a solução usada, por exemplo, no coletor por contagem de referências da linguagem Modula-2+, segundo relatado por DeTreville [12]. Infelizmente é uma solução que dificulta o escalonamento para um maior número de processadores: intuitivamente, quanto maior o número de processadores executando concorrentemente, maior será a contenção para obter o *mutex* para alterações de ponteiros, e portanto mais tempo será gasto com sincronização; com um número suficiente de processadores, pode-se imaginar que a sincronização chega a tomar um tempo comparável ou mesmo maior que o tempo gasto em computação útil. O relatório técnico de DeTreville [12] cita uma experiência deste tipo, na qual um coletor que utilizava somente contagem de referências foi descartado por ter desempenho insatisfatório.

Por último, a manutenção da lista livre inclui mais duas condições de corrida possíveis: quando é preciso alocar uma nova célula, e portanto retira-la da lista livre; e quando é preciso devolver uma célula para a lista, no momento da liberação. Alterações concorrentes sem exclusão mútua podem corromper a lista livre, inviabilizando o uso continuado do sistema.

Nas seções que seguem serão apresentadas e analisadas as arquiteturas propostas para implementar a contagem de referências em sistemas multiprocessados, e para cada uma delas será observado como esses problemas são resolvidos, e quão satisfatórias são as soluções empregadas.

4.2.2 Coletor concorrente com um mutador e um coletor

Esta seção apresenta a primeira arquitetura para contagem de referências concorrente. Assim como no artigo de Lins [31], essa primeira proposta inicialmente prevê apenas um mutador e um coletor, executando em processadores diferentes. Na arquitetura de Lins mesmo o mutador realiza algumas tarefas de coleta, como o incremento nos contadores de referências, mas também realiza computação útil, mantendo o forte acoplamento típico da técnica de contagem de referências. Já a arquitetura proposta aqui estabelece uma maior separação entre as tarefas do mutador e coletor, embora seja impossível dispensar completamente a coordenação entre eles. Os algoritmos usados para gerenciamento de memória nas arquiteturas concorrentes apresentadas aqui são baseados na contagem de referências cíclicas, como visto no Capítulo 3.

A coordenação entre coletor e mutador se dá por meio de três estruturas de dados: a lista livre – tratada como uma fila – e duas filas que registram alterações nas referências, chamadas de *fila de incrementos* e *fila de decrementos*. O mutador não altera diretamente os contadores de referência, mas inclui nestas duas filas pedidos de incremento e decremento do contador, conforme uma referência tenha sido criada ou destruída. Desta forma, só o coletor altera os contadores de referências, evitando quaisquer condições de corrida ou problemas de sincronização, e eliminando a necessidade de usar semáforos. Esta organização é similar ao *arquivo de transações* do coletor de Deutsch e Bobrow [13], mas com uma representação bastante eficiente e na memória principal. Com estas três estruturas em forma de fila, o mutador tem acesso apenas a uma das extremidades das filas, e o coletor acessa a outra extremidade; assim, para uma dada fila, apenas um dos dois processadores inclui itens, enquanto o outro apenas exclui itens da mesma. O mutador retira células da lista livre para utiliza-las, e inclui pedidos de incremento e decremento de contadores de referências; o coletor retira e processa pedidos de incremento e decremento, e adiciona células liberadas à lista livre. A idéia geral desta arquitetura está expressa na Figura 4.1. Como é usual inserir itens no final de uma fila, e retirá-los no início, o mutador tem acesso ao final das filas de incremento e decremento (`bot-inc-queue` e `bot-dec-queue`, respectivamente), e ao topo da lista livre (`top-free-list`). Já o coletor tem acesso ao topo das filas de incremento e decremento (`top-inc-queue` e `top-dec-queue`), e ao final da lista livre (`bot-free-list`). Para implementar estas filas de maneira concorrente, pode-se usar algum dos algoritmos conhecidos para filas *lock-free*, que são especialmente simples no caso de apenas um produtor e um consumidor [45].

A decisão de usar duas filas para os pedidos de alteração nos contadores de referências foi motivada por questões de eficiência na representação: desta forma, um item em uma das filas é apenas o endereço da célula, já que a operação requerida no contador – incrementar ou decrementar – fica ímplicita na própria fila onde o item está localizado. Em geral, o tamanho de um endereço é igual ao tamanho da palavra, então cada item nessas filas ocupa exatamente uma palavra na memória. Caso fosse desejado usar uma fila apenas para todos os pedidos, seria necessário usar pelo menos um bit para especificar a operação de ajuste necessária, o que impediria ou, no mínimo, dificultaria a representação de cada item individual em uma palavra. Além disso, essa organização permite processar os incrementos antes dos decrementos, o que é importante para garantir a corretude do programa, como será visto adiante.

O processador $P1$ é o mutador, e $P2$ o coletor. O mutador está encarregado das alterações

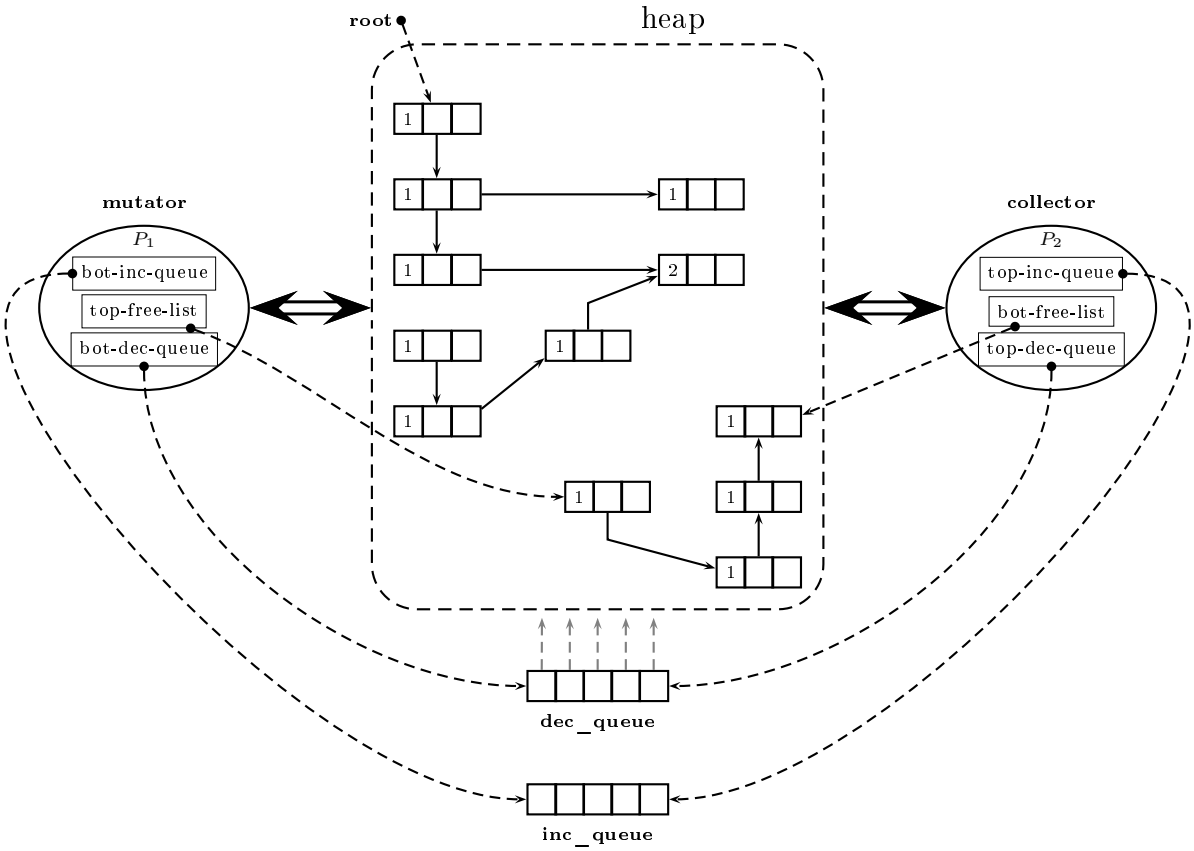


Figura 4.1: *Arquitetura concorrente proposta para um coletor e um mutador*

no grafo da memória – que decorrem das operações de computação útil – juntamente com a alocação de novas células da lista livre e a inclusão de pedidos para incremento e decremento de referências. O coletor, por sua vez, tem como tarefa o processamento dos pedidos de incremento e decremento de referências, alterando o valor dos contadores envolvidos, e a liberação efetiva das células, devolvendo-as para a lista livre. Estas operações são descritas mais formalmente abaixo.

Operações do mutador As operações do mutador, com relação ao gerenciamento de memória, são:

▷ `New` – aloca novas células

▷ `Del` – gera um pedido para remoção de uma referência

▷ `Update` – altera uma referência

Em alguns algoritmos de contagem de referências se inclui também uma operação `Copy` que copia ou duplica uma referência, mas esta não é necessária aqui, sendo um caso particular da rotina `Update`.

Como o algoritmo utilizado como base é o de contagem de referências cíclicas, as células incluem não só um contador de referências mas também uma cor, para ser usada pelo coletor; isso traz conseqüências para a sincronização que são similares às do contador de referências, como será explicado adiante. A coordenação da lista livre e das filas de incremento e decremento é feita guardando o endereço para uma das extremidades de cada uma delas no mutador, enquanto a outra é guardada no coletor, através dos pares de registradores como `top-inc-queue` e `bot-inc-queue`.

A rotina para `New` aparece na Figura 4.2. Primeiro verifica-se se a lista livre tem células disponíveis; se for o caso, a célula no topo da lista livre é obtida e retirada da lista livre. Aqui assume-se que a lista livre está organizada como uma lista duplamente encadeada e que o campo `prev` de uma célula na fila aponta para a célula anterior na fila; esta é uma representação comum para filas, e é aqui utilizada nas três filas do sistema. Caso não existam células na lista livre, o mutador entra em espera ocupada (*busy waiting*) até que o coletor devolva ao menos uma célula para a lista livre.

```
New() =
   if top-free-list != nil then
      newcell = top-free-list
      top-free-list = top-free-list.prev
   else
      newcell = New()
   return newcell
```

Figura 4.2: *Rotina para alocação de novas células*

Quando uma referência deve ser removida, é chamada a rotina `Del`, mostrada na Figura 4.3. A rotina simplesmente cria um pedido para incluir a célula S – para a qual uma referência está sendo removida – na fila de decrementos, chamando a rotina `add_decrement`; esta é encarregada de adicionar um item na fila de decrementos, usando o campo `next` para encadeamento.

Por fim, a rotina `Update` é utilizada quando é necessário alterar um ponteiro R para apontar para uma célula S. A Figura 4.4 mostra o algoritmo para `Update`. Primeiro, remove-

```
Del(S) =
   add_decrement(S)

add_decrement(S) =
   S.next = top-del-queue
   top-del-queue = S
```

Figura 4.3: *Algoritmo para remoção de referências*

se uma referência para a célula apontada por R (representada aqui por `*R`); em seguida, insere-se um pedido para incrementar o contador de referência de S – através da chamada a `add_increment` – e, por fim, atribui-se o endereço de S para o ponteiro R. Aqui é importante notar que um ponteiro é alterado, uma operação passível de sincronização, como visto na seção anterior. Entretanto, como só há um mutador, não é necessário sincronizar as alterações de ponteiros, pois o coletor nunca muda o valor de um ponteiro em uma célula ativa. Por fim, `add_increment` apenas adiciona o endereço da célula em questão na fila de incrementos, de forma similar ao processo de inclusão de itens em todas as filas.

```
Update(R, S) =
   Del(*R)
   add_increment(S)
   *R = S

add_increment(S) =
   S.next = top-inc-queue
   top-inc-queue = S
```

Figura 4.4: *Alteração de referências na arquitetura proposta*

Operações do coletor Como dito anteriormente, a função do coletor é processar os pedidos de alteração no contador de referências, e liberar as células que se tornem inativas – ou seja, cujo contador chega ao valor zero. Como o algoritmo para varredura local em busca de ciclos opera durante a remoção de referências, isso conseqüentemente é uma função do coletor também.

A operação principal do coletor é chamada de `Process_queues`, que roda continuamente na *thread* do processador $P2$ como mostrado na Figura 4.5; sua função principal é processar os pedidos de incremento e decremento nas filas. O primeiro passo da rotina é examinar a

fila de incremento: cada incremento encontrado na fila é retirado desta e causa o incremento do contador da célula apontada, e a alteração de sua cor para verde; isso é feito até que a fila esteja vazia. Só então `Process_queues` passa a tratar dos decrementos. Isso é feito para garantir que o sistema estará sempre à frente no processamento dos incrementos em relação aos decrementos, o que evita que uma célula com apenas uma referência, cujo contador vai ser incrementado e decrementado logo em seguida, seja liberada antes que o incremento seja visto, por causa de diferenças no escalonamento dos processadores. É importante notar que processar os incrementos à frente dos decrementos nunca deixará o sistema em algum estado inconsistente: uma célula que se torna inatingível nunca será incrementada além de zero, e será eventualmente liberada; e como os contadores são sempre incrementados primeiro, é obviamente impossível que uma célula ativa seja encarada como lixo. Supondo um programa que gere incrementos e decrementos aproximadamente na mesma taxa, o que é uma suposição bastante razoável, é de se esperar que o coletor processe primeiro todos os incrementos de um grupo, e depois os decrementos, mantendo sempre a corretude do sistema sem deixar de processar os decrementos por muito tempo. O processamento da fila de decrementos é mais simples: a rotina apenas verifica se a fila está vazia e, caso contrário, retira uma célula do fim da fila e chama a rotina `Rec_del` para efetivamente remover a referência. Para tirar um item das filas de incremento e decremento, utiliza-se as rotinas auxiliares `get_increment` e `get_decrement`, respectivamente; estas apenas retornam a célula no topo da fila correspondente, e alteram o topo para apontar para a célula anterior.

Se não houver pedidos de incremento nem de decremento, o coletor ficaria ocioso, e para aproveitar esse tempo ocioso a rotina `Process_queues` chama `scan_status_analyser` para realizar varreduras locais em busca de ciclos de células inativas. Caso a varredura do *status analyser* ocorra com mais freqüencia do que desejado, pode ser adicionado um contador para só chamar `scan_status_analyser` depois que as filas de incremento e decremento forem encontradas vazias um certo número de vezes. Esse tipo de ajuste de parâmetros pode ser feito pelo programador do sistema, que terá mais conhecimento para achar o valor mais adequado.

`Rec_del` é a rotina para liberação recursiva de células inativas, mostrada na Figura 4.6. Ela funciona de uma forma similar à rotina `Delete` do algoritmo seqüencial para contagem de referências cíclicas (ver Figura 3.3): se a célula tiver contador de referências com valor 1, ela será liberada, mudando sua cor para verde, e ligando-a à lista livre, antes disso eliminando

```
Process_queues() =
   while bot-inc-queue != nil
      S = get_increment()
      S.rc = S.rc + 1
      S.color = green
   if bot-dec-queue != nil then
      S = get_decrement()
      Rec_del(S)
   else
      scan_status_analyser()
   Process_queue()

get_increment() =
   C = top-inc-queue
   top-inc-queue = top-inc-queue.prev

get_decrement() =
   C = top-dec-queue
   top-dec-queue = top-dec-queue.prev
```

Figura 4.5: *Processamento das filas de incremento e decremento*

recursivamente as suas referências. Caso o contador tenha valor maior que 1, isso pode indicar que um ciclo de lixo foi gerado, e então a célula é adicionada ao *status analyser*, se já não estiver nele.

A varredura local em busca de ciclos é feita pela rotina `scan_status_analyser`, que é igual à do caso seqüencial (Figura 3.4). `Mark_red`, que é chamada por `scan_status_analyser`, também é quase idêntica, mas por uma questão de sincronização há uma mudança no decremento do contador de referências. A rotina alterada está na Figura 4.7, e muda apenas a linha onde ocorre o decremento dos contadores de referências. O mesmo acontece com `Scan_green`, que só é alterada na linha onde o contador de referências é incrementado, como visto na Figura 4.8. Por último, `Collect` permanece igual ao caso seqüencial (Figura 3.7), embora altere também a contagem de referência e cor das células. O motivo é visto adiante, na análise da sincronização desta arquitetura.

Análise A arquitetura proposta difere da de Lins principalmente na forma de tratar a interferência entre coletor e mutador: Lins recomenda tratar o problema através do uso de sincronização explícita, utilizando semáforos, embora isso não seja detalhado na sua pro-

```
Rec_del(S) =
   if S.rc == 1 then
      S.color = green
      for T in S.children do
         Rec_del(T)
      bot-free-list.next = S
      bot-free-list = S
   else
      if S.color != black then
         S.color = black
         add_to_status_analyser(S)
```

Figura 4.6: *Liberação recursiva*

```
Mark_red(S) =
   if S.color != red then
      S.color = red
      for T in S.children do
         decrement_rc(T)
      for T in S.children do
         if T.color != red then
            Mark_red(T)
         if T.rc > 0 && T not in status-analyser then
            add_to_status_analyser(T)
```

Figura 4.7: *Rotina para marcar células de vermelho*

```
Scan_green(S) =
   S.color := green
   for T in S.children do
      increment_rc(T)
      if T.color != green then
         scan_green(T)
```

Figura 4.8: *Rotina para varrer células pintadas de verde*

posta; já a arquitetura apresentada acima utiliza as técnicas de *variáveis disjuntas* – pois coletor e mutador não concorrem no acesso de variáveis na memória – e *enfraquecimento de asserções* – pois o invariante da contagem de referências é enfraquecido – para lidar com a interferência [2], o que evita o uso de semáforos e melhora a eficiência do algoritmo concorrente resultante. De fato, o sistema de gerenciamento de memória não utiliza nenhuma operação de sincronização, em comparação com a versão clássica do algoritmo concorrente de contagem de referências [12], que precisa obter um semáforo global associado a todas as *threads*. Considera-se atualmente que o uso de semáforos afeta negativamente o desempenho do sistema a ponto de tornar-se proibitivo [11, 12, 24]; há também o problema de usar apenas um semáforo para todas as *threads*, mas isto não afeta a arquitetura em questão pois está previsto apenas um mutador.

Com relação ao aproveitamento dos recursos computacionais, é plausível que o processador do coletor fique ocioso, se as filas de incremento e decremento estiverem vazias e não houver células no *status analyser*; entretanto, esse problema concerne mais ao escalonador do sistema operacional, que pode utilizar o processador alocado para o coletor em outras tarefas.

Uma otimização interessante que é possibilitada por essa arquitetura é que é identificar alterações redundantes nos contadores de referências. Por exemplo, se houver um pedido de incremento para uma célula C na fila de incremento, e um pedido de decremento para a mesma célula na fila de decremento, o valor do contador de C não deve ser alterado. Se ao invés de processar os pedidos um de cada vez a rotina `Process_queue` analisar um grupo de pedidos nas duas filas, é possível cancelar pedidos e mesmo alterar os contadores de forma mais eficiente. Uma sugestão para implementar isso é obter um lote de itens nas duas filas e processá-los conjuntamente, usando uma tabela *hash* cuja chave é o endereço da célula e o valor associado é o valor final do contador de referência; cada pedido processado altera o valor associado na tabela hash, sem precisar alterar a célula em si. Ao final de um lote, pode-se enumerar os itens na tabela e mudar de uma vez os contadores de referência para seus valores finais após o lote. Isso evitará alterações redundantes e pode ser mais eficiente mesmo sem a presença de redundância, pois a tabela *hash* deverá apresentar melhor localidade de referência no coletor. Outras análises são possíveis baseadas nas filas de incremento e decremento, como evitar de incluir no *status analyser* uma célula que será, em seguida, liberada; mas o estudo dessas possibilidades é deixado como uma sugestão para trabalhos futuros.

4.2.3 Coletor concorrente com vários mutadores e um coletor

A extensão da arquitetura proposta para vários mutadores (mas ainda um coletor) é mostrada na Figura 4.9. A única diferença para o caso de um mutador é que deve existir alguma sincronização para o acesso à extremidade das filas que ficam acessíveis aos mutadores, então estes compartilham referências para os registradores globais `bot-inc-queue`, `bot-dec-queue` e `top-free-list`. Uma forma eficiente de realizar a sincronização nestes registradores é utilizar operações do tipo *compare-and-swap* para atualizar o valor dos mesmos.

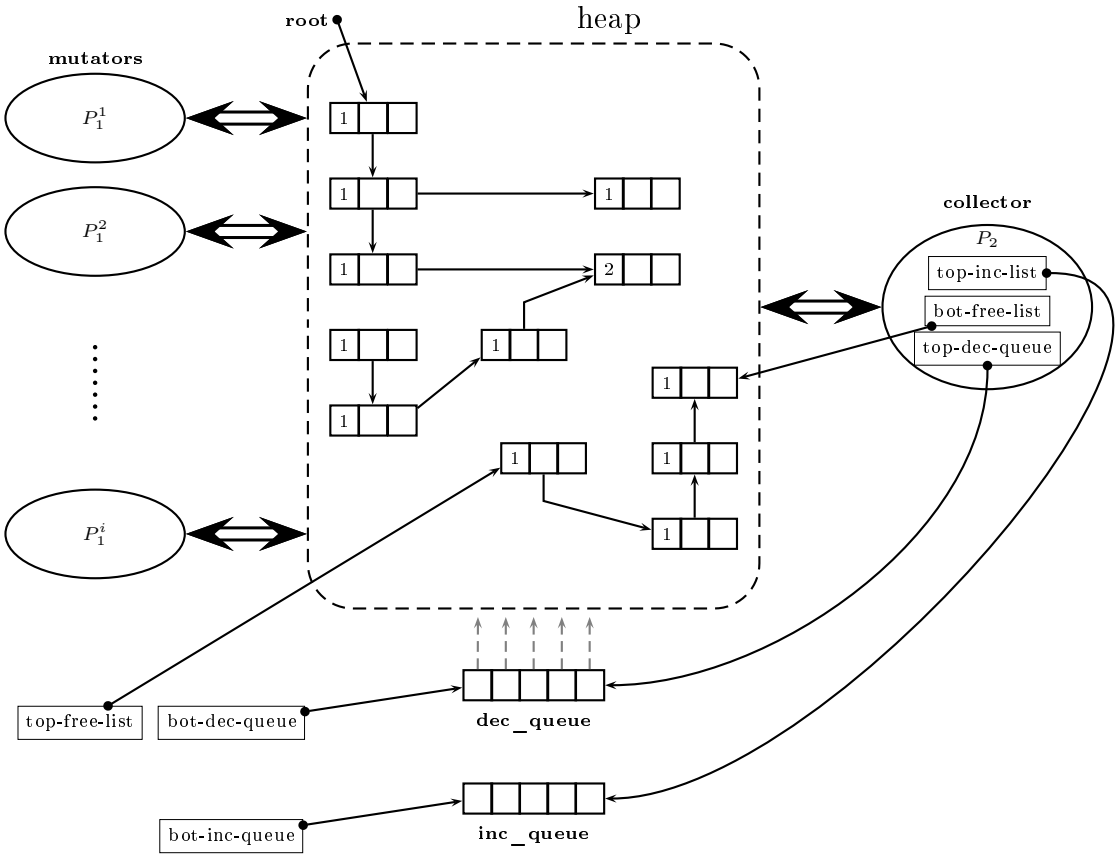


Figura 4.9: *Arquitetura concorrente proposta para um coletor e vários mutadores*

Operações *compare-and-swap* são implementadas em *hardware* na maioria das arquiteturas atuais. Essas operações só alteram uma localização na memória para um novo valor n se o valor atual, v for igual ao esperado pela *thread*. Ou seja, fica garantido que a *thread* lê o valor atual e escreve o novo valor, tudo isso de forma atômica. Com isso, cada *thread* só competirá com outras *threads* que tentarem alterar o mesmo valor, sem a necessidade de obter semáforos

de qualquer tipo. Além disso, operações do tipo *compare-and-swap* são mais eficientes do que esperar por semáforos, o que pode ser um impacto significativo na arquitetura com vários mutadores, na qual a operação das filas compartilhadas tende a se tornar freqüente.

Entretanto, um problema potencial é a alteração dos valores dos ponteiros em `Update`, pois vários mutadores podem realizar alterações no mesmo ponteiro simultaneamente, deixando-o com um valor inconsistente ao final. Por este motivo, a rotina `Update` é alterada para usar explicitamente uma operação *compare-and-swap* quando é necessário alterar o valor de um ponteiro, como mostrado na Figura 4.10.

```
Update(R, S) =
   Del(*R)
   add_increment(S)
   compare_and_swap(*R, S)
```

Figura 4.10: *Alteração de referências na arquitetura com vários mutadores*

Análise Em contraste com o caso para um coletor e um mutador, onde o sistema de gerenciamento de memória não necessita de nenhuma sincronização, no caso para vários mutadores a sincronização necessária passa a ser algumas operações *compare-and-swap*, que ainda são mais eficientes do que semáforos. As propostas recentes para utilizar contagem de referências em coletores concorrentes sempre procuram evitar o uso de semáforos [11, 24].

Com relação ao uso dos recursos computacionais, neste caso o coletor deve servir mais de um mutador, e portanto deve ficar mais ocupado. É de se esperar, portanto, que é ainda menos provável que o processador do coletor fique ocioso.

Uma consideração interessante quanto às possibilidades de escalonamento da arquitetura proposta é com relação à sincronização necessária para as filas. Mesmo usando uma operação de sincronização eficiente, os topos das filas são compartilhados entre todos os mutadores; enquanto isso, espera-se que a competição para alterar ponteiros específicos fique restrita a uma parcela dos mutadores, se o número deles for suficientemente grande. Isso indica que esta arquitetura não escalona bem para grandes números de mutadores, pois quanto maior o número deles, maior a competição pelos topos das filas, e a contenção deve crescer rapidamente com o número de mutadores, pois alterações de ponteiros e alocações são freqüentes em sistemas atuais, sejam orientados a objetos ou funcionais. Sem ter como experimentar di-

retamente com computadores com 8 ou mais processadores, é difícil determinar até que ponto essa arquitetura permanece válida. Uma proposta para amenizar esse problema é ter várias filas separadas, numa proporção de um conjunto de filas para cada N mutadores; o valor de N deve ser determinado experimentalmente. No limite, poderia-se ter um conjunto de filas para cada mutador, mas isso provavelmente é um super-dimensionamento. O problema merece estudo, mas os prospectos são promissores: a experiência de Flood *et al.* [16] com coletores paralelos na máquina virtual Java mostrou que tornar a alocação concorrente resultou em um ganho de desempenho de 30% no tempo total de execução.

4.2.4 Coletor concorrente e paralelo

Lins [31] sugere algumas possibilidades para extender sua arquitetura para vários processadores coletores, tornando o conjunto um coletor concorrente e paralelo. A principal é que cada coletor tenha um *status analyser* local, para que várias varreduras locais possam ser executadas em paralelo; cada coletor, ao identificar uma célula candidata para o *status analyser*, adiciona-a à sua versão local da estrutura.

As possibilidades de extensão para a arquitetura proposta seguem idéias similares, mas levando também em consideração o problema do escalonamento mencionado na análise do coletor com vários mutadores e um coletor: replicar não só o *status analyser*, mas também as três filas – incrementos, decrementos e a lista livre. Cada conjunto de estruturas de gerenciamento de memória seria tratada apenas por um coletor, e estaria disponível para um conjunto de mutadores. Na prática, então, haveria um coletor – com todas as estruturas de dados associadas – para N mutadores, sendo o valor desse N determinado experimentalmente. A relação entre mutadores e coletores poderia ser mais um parâmetro ajustável pelo programador, pois pode depender de características específicas do programa. Para melhorar o balanceamento de carga, pode-se incluir formas de migrar itens nas filas e no *status analyser* entre coletores diferentes, talvez usando alguma forma de *work stealing* [16].

Entretanto, o estudo dessa modalidade não será detalhado aqui, ficando como possibilidade para trabalhos futuros.

4.2.5 Corretude

Os coletores de lixo paralelos e concorrentes apresentados na literatura [11, 16, 24, 31] normalmente não possuem provas formais de corretude, mas apenas argumentos informais.

Embora os argumentos sejam convincentes, é importante lembrar que mesmo profissionais competentes podem ignorar erros e contra-exemplos para provas que são feitas informalmente, como relatado por Dijkstra *et al.* [14]. Aqui se apresenta, então, apenas um argumento informal sobre a corretude das arquiteturas propostas. Embora trabalhos recentes incluam provas formais para sistemas de gerenciamento da memória de sistemas reais [6, 7], as dificuldades para provar formalmente sistemas concorrentes e paralelos são reconhecidamente muito maiores, e não é do conhecimento do autor que nenhuma prova formal para sistemas práticos – e não meramente teóricos – tenha sido publicada.

A arquitetura proposta com um mutador e um coletor (Seção 4.2.2) executa sem a necessidade de usar sincronização para evitar interferência entre os dois; isso é possível porque coletor e mutador acessam variáveis disjuntas, e a invariante da contagem de referências é enfraquecida. O funcionamento deste algoritmo concorrente é, portanto, mais simples de entender e formalizar que a versão anterior de Lins. Este é um efeito colateral interessante de minimizar o uso de sincronização explícita: o sistema resultante se torna mais simples de entender, evitando condições de corrida sutis que podem permitir caminhos de execução pouco intuitivos. Os dois aspectos de corretude para um coletor de lixo são que ele deve identificar todas as células inativas, e que nenhuma célula ativa pode ser interpretada como lixo; o primeiro requerimento algumas vezes é relaxado, no que se costuma chamar de *coletores conservativos*. No caso da arquitetura proposta, processar os incrementos sempre de maneira adiantada em relação aos decrementos ajuda a garantir que uma célula ativa não será liberada por uma alteração na ordem de processamento dos pedidos de incremento e decremento; esta é a única situação em que uma célula ativa poderia ser interpretada como lixo. Por outro lado, todas as células inativas são eventualmente liberadas, pois após se tornarem inativas não serão mais criadas referências para ela, e eventualmente os pedidos de decremento serão processados até que seu contador chegue a zero, causando sua liberação. Só o coletor altera as contagens de referência, então estas nunca estarão em um estado inconsistente; mas note-se que a invariante da contagem de referências é enfraquecida: em um dado momento, o contador de referências de uma célula C não é necessariamente igual ao número de células que apontam para C, se houver pedidos de incremento e decremento do contador em C que não tenham sido processados ainda. Mas isso não altera a corretude do coletor, que executa operações de acordo com os pedidos de incremento e decremento processados; estas operações podem ser adiadas com relação ao que ocorreriam em um coletor sequencial, mas sempre serão realizadas.

Para o coletor com vários mutadores e um coletor (Seção 4.2.3) o raciocínio é similar, devendo ser levado em conta agora as atualizações de ponteiros na rotina `Update` e as mudanças no topo das três filas de controle. Como estas alterações são realizadas atomicamente, com o uso de operações de *compare-and-swap*, não ocorrerão valores inconsistentes no programa. De resto, o coletor único trabalha da mesma forma, e não se vislumbram aspectos que possam afetar sua corretude.

4.3 Trabalhos relacionados

A idéia de registrar pedidos de alteração nos contadores de referências, ao invés de efetuar essas mudanças diretamente, é similar ao arquivo de transações no sistema de contagem de referências de Deutsch e Bobrow [13]; embora neste trabalho os autores não tenham tido como objetivo projetar um sistema de gerenciamento de memória concorrente, eles observam que a organização baseada em um arquivo de transações é adeqüada para sistemas concorrentes. O coletor concorrente por contagem de referências da linguagem Modula-2+ [12] pode ser considerado uma tentativa de aplicar as idéias de Deutsch e Bobrow neste sentido. Entretanto, esta solução foi considerada proibitivamente ineficiente pelo impacto de desempenho que era imposto em cada operação com ponteiros; o uso de um semáforo global permitia apenas uma operação com ponteiro por vez em todo o programa, independente do número de *threads* concorrentes.

Nos últimos anos a pesquisa em gerenciamento de memória concorrente e paralelo tem se concentrado em minimizar a necessidade de sincronização imposta pelo coletor de lixo. Herhily e Moss [19] apresentaram um dos primeiros algoritmos para coleta de lixo *lock-free*, ou seja, que não usa nenhuma operação que envolva obter semáforos. Este algoritmo possui sincronização leve, mas requer que várias versões de um mesmo objeto sejam guardadas na memória – sempre que algum campo de um objeto é alterado, a versão anterior deve permanecer acessível, e é criada uma nova versão do objeto com o novo valor do campo; isso é similar à idéia de atualização não-destrutiva da programação funcional. Na prática, há grandes custos tanto de tempo quanto de espaço para manter as versões de todos os objetos do programa, o que torna este algoritmo indesejável em situações reais.

Levanoni e Petrank [24] sugeriram uma arquitetura baseada no trabalho de Deutsch e Bobrow, usando pouquíssima sincronização – não é necessário nem obter semáforos, nem utilizar operações do tipo *compare-and-swap*. Mas, para chegar a tal resultado, o algoritmo

se tornou bastante complexo, principalmente pela necessidade de gerenciar o uso de retratos (*snapshots*) da memória que são obtidos em tempos diferentes por cada *thread* – isto é chamado no artigo de uma *sliding view*. Não fica claro, pelas medições mostradas, se a eficiência na sincronização compensa a complexidade do algoritmo final; mas é preciso lembrar que além da eficiência deve-se considerar outros custos, como a manutenção dos programas. Programas complexos têm maior custo de manutenção.

Um outro trabalho que propõe um algoritmo concorrente para contagem de referências é o de Detlefs *et al.* [11]; este necessita de operações DCAS (*double compare-and-swap*), que garantem que duas localizações na memória serão alteradas atomicamente. Tais operações são necessárias para atualizar os contadores de referências de duas células R e S quando uma referência a R é alterada para apontar para S: a primeira tem seu contador decrementado, e a segunda recebe um incremento no contador. Os próprios autores observam que esta operação dupla não é fácil de ser implementada em arquiteturas atuais, o que torna o algoritmo de difícil aplicabilidade. Embora operações do tipo *compare-and-swap* sejam encontradas hoje em praticamente todas as arquiteturas, a versão dupla não existe em nenhuma delas.

capítulo 5

Testes e Resultados

No Capítulo 4 foram descritas e propostas algumas arquiteturas para a implementar o gerenciamento da memória por contagem de referências em sistemas multiprocessados. A maior vantagem destas arquiteturas mostradas é a pequena necessidade de sincronização entre as *threads*, tornando-as eficientes e mais facilmente escalonáveis para computadores multiprocessados com vários processadores.

Neste capítulo são apresentados os resultados de alguns testes de desempenho realizados em uma implementação de uma das arquiteturas propostas: a que prevê apenas um mutador e um coletor. Para isto, os testes foram realizados em computadores com dois processadores, para que o mutador e o coletor sejam executados concorrentemente. Esta implementação foi comparada com uma versão seqüencial do algoritmo de contagem de referências cíclicas, como mostrado no Capítulo 3.

Embora os testes realizados não caracterizem de forma definitiva o desempenho relativo da versão concorrente com relação à versão seqüencial, os resultados indicam claramente uma tendência de maior eficiência na versão concorrente, como era de se esperar. Outras comparações e análises de desempenho são sugeridas como trabalhos futuros no Capítulo 6.

5.1 Plataforma de testes

Para realizar os testes foi implementado um compilador para uma linguagem funcional simples. A implementação de todo um compilador foi julgada necessária para garantir a execução concorrente em multiprocessadores, tendo em vista que vários compiladores em uso

atualmente, apesar de gerarem código com múltiplas *threads*, não garantem que essas *threads* serão executadas pelos diferentes processadores em um computador multiprocessado.

O compilador implementado gera código para a máquina-G [41] e então traduz este código para código nativo para os processadores Intel IA-32. O Apêndice A descreve a linguagem utilizada e o compilador em maiores detalhes. Para os propósitos deste capítulo, é importante que o compilador gera código nativo e executa as operações de uma máquina de redução de grafos; as atividades de redução no grafo são responsáveis por gerar células inativas, criando a necessidade de um coletor de lixo. Além disso, a implementação de funções recursivas na redução de grafos cria ciclos, que não podem ser recuperados com o algoritmo tradicional de contagem de referências.

Os programas de teste (ver Apêndice B) foram executados em computadores com dois processadores Athlon MP de $1.6GHz$ e 512Mb de memória RAM, utilizando-se três versões do sistema de gerenciamento da memória: uma versão com o algoritmo seqüencial, uma com o algoritmo concorrente para um mutador e um coletor, e uma última sem nenhuma recuperação de objetos inativos. Esses programas foram selecionados para realizar medições de desempenho por causarem a geração de muitos ciclos durante a execução; como a máquina G implementa funções recursivas através do combinador Y, que forma um ciclo no grafo da expressão a ser reduzida, a maioria dos programas utilizados geram muitas chamadas recursivas às funções principais. Também foram testados programas que geram grandes estruturas de listas que não são recursivas nem cíclicas. As cargas de trabalho utilizadas para cada teste podem ser vistas no texto completo dos programas, incluído no Apêndice B.

5.2 Resultados e análises

Cada um dos seis programas foi executado 10 vezes por versão, medindo-se o tempo de cada execução, e calculou-se a média dos 10 tempos obtidos. Os resultados são mostrados na Tabela 5.1, na qual a coluna denominada *sem-rc* indica os tempos obtidos para os programas sem o uso de gerenciamento automático da memória, a coluna *rc-seq* mostra os tempos obtidos com o algoritmo de contagem de referências seqüencial, e a coluna *rc-conc* os tempos obtidos para o algoritmo concorrente. Todos os tempos indicados estão em segundos. As Tabelas 5.2 e 5.3 apresentam, como dados complementares, os perfis de execução dos testes, tanto na versão seqüencial (Tabela 5.2) quanto na versão concorrente do algoritmo de coleta de lixo (Tabela 5.3). Nestas tabelas, a coluna `alloc` indica o número total de células alocadas durante

Tabela 5.1: *Tempos de execução para os programas de teste, em segundos*

Teste	*sem-rc*	*rc-seq*	*rc-conc*
acker	0.0200	0.0351	0.0250
conctwice	0.0030	0.0050	0.0038
fiblista	2.2733	3.0812	2.7320
recfat	0.0521	0.0872	0.0702
somamap	0.0234	0.0473	0.0298
somatorio	0.0133	0.0274	0.0208
tak	12.731	22.146	17.764
queens	6.8023	11.904	9.1091

Tabela 5.2: *Perfil de execução dos testes para o algoritmo seqüencial*

Teste	`alloc`	`scan_sa`	`mark_red`	`scan_green`	`collect`	tempo
acker	253175	4013	40574	40127	447	0,0351
conctwice	42834	521	5468	5406	62	0,0050
fiblista	14837640	27892	857421	836317	21104	3,0812
recfat	335959	4985	49872	49338	534	0,0872
somamap	94309	1356	10521	10409	112	0,0473
somatorio	35298	499	4987	4933	54	0,0274

o programa; `scan_sa` mostra o número de chamadas à operação `scan_status_analyzer`; `Mark_red`, `scan_green` e `collect` denotam o número de chamadas às funções de mesmo nome; e a última coluna mostra o tempo total de execução do teste, que é o mesmo observado na Tabela 5.1, colunas *rc-seq* e *rc-conc*.

Para analisar melhor as diferenças nos tempos, a Tabela 5.4 mostra as diferenças relativas entre as três versões do mesmo programa, para identificar o impacto das alterações no gerenciamento de memória.

Tabela 5.3: *Perfil de execução dos testes para o algoritmo concorrente*

Teste	`alloc`	`scan_sa`	`mark_red`	`scan_green`	`collect`	tempo
acker	253175	9304	57966	57434	532	0,0250
conctwice	42834	897	7210	7127	83	0,0038
fiblista	14837640	35112	956874	935527	21347	2,7320
recfat	335959	11421	112663	112075	588	0,0702
somamap	94309	1647	11896	11773	123	0,0298
somatorio	35298	532	6052	5980	72	0,0208

Tabela 5.4: *Diferença relativa entre os tempos de execução das versões diferentes*

Teste	*rc-seq/sem-rc*	*rc-conc/rc-seq*	*rc-conc/sem-rc*
acker	0.7550	0.2877	0.2500
conctwice	0.6667	0.2400	0.2667
fiblista	0.6633	0.2537	0.2414
recfat	0.6737	0.1950	0.3474
somamap	1.0209	0.3698	0.2735
somatorio	1.0602	0.2409	0.5639
tak	0.7395	0.1978	0.3953
queens	0.7499	0.2349	0.3391

Os números na coluna *rc-seq/sem-rc* da Tabela 5.4 foram calculados segundo a fórmula

$$\frac{t_{rc-seq} - t_{sem-rc}}{t_{sem-rc}}$$

onde t_{rc-seq} é o tempo de execução do programa correspondente usando o algoritmo seqüencial, e t_{sem-rc} é o tempo para a versão do programa sem gerenciamento automático de memória, e os tempos são obtidos para o programa em questão a partir da Tabela 5.1. Ou seja, esta coluna mostra o quanto o algoritmo seqüencial é mais lento em relação à versão sem gerenciamento automático da memória.

De forma similar, os números na coluna *rc-conc/rc-seq* foram calculados pela fórmula

$$\frac{t_{rc-seq} - t_{rc-conc}}{t_{rc-seq}}$$

e portanto indicam o quanto a versão concorrente é mais rápida que a versão seqüencial do algoritmo. Por fim, a última coluna foi calculada com o uso da fórmula

$$\frac{t_{rc-conc} - t_{sem-rc}}{t_{sem-rc}}$$

e indicam o quanto a implementação concorrente do algoritmo é mais lenta do que executar o programa sem nenhum gerenciamento de memória.

A primeira observação é com relação ao impacto do algoritmo de contagem de referências seqüencial no desempenho do programa: a diferença relativa variou de 66.33% a 106%. Embora bastante elevada, a diferença pode ser justificada com o fato dos programas testados serem intensivos em computação, não realizando praticamente nenhuma operação de entrada e saída. Todos os programas alocam uma grande quantidade de células e geram vários ciclos, por meio de chamadas recursivas. Isso indica que o sistema de gerenciamento de memória é exigido praticamente durante toda a execução do programa, afetando em muito os números de

desempenho. Em programas que utilizam mais a entrada e saída, a diferença deve se mostrar bem menor.

Outro fato importante para entender esses resultados é que os programas testados são curtos, e muitos programas inicialmente preparados para as medidas de desempenho não puderam ser utilizados, pois a implementação compilada resultou em tempos de execução tão pequenos que as diferenças seriam insignificantes. Os programas cujos resultados constam na Tabela 5.1, entretanto, mostram uma tendência consistente, o que indica que uma tendência real foi observada. Também é importante notar que a versão sem gerenciamento de memória simplesmente ignora os vazamentos de memória; de certa forma, os programas nessa versão podem ser considerados efetivamente incorretos, já que a memória não é recuperada de forma alguma. Mesmo assim, a comparação foi realizada para ter uma idéia do impacto do gerenciamento automático da memória no desempenho geral.

Com relação aos tempos do algoritmo concorrente de contagem de referências cíclicas, pode-se ver na Tabela 5.4 que esta versão foi de 19 a 36 por cento mais rápida que a versão seqüencial e de 25 a 56 por cento mais lenta que a versão sem gerenciamento da memória. Os resultados obtidos estão dentro de uma faixa coerente, nos dois casos, e indicam claramente uma vantagem de cerca de 20% do algoritmo concorrente em relação à sua versão seqüencial, um ganho promissor. Isto ainda pode ser associado a outras otimizações no algoritmo concorrente, como comentado no Capítulo 4.

capítulo 6

Conclusões e Trabalhos Futuros

Este capítulo, o último do presente trabalho, apresenta um balanço do trabalho realizado e dos resultados obtidos na pesquisa que culminou nesta dissertação. Sugestões para trabalhos futuros que podem ser realizados como continuações do estudo também são apresentadas.

6.1 Conclusões

A contagem de referências é uma das técnicas mais utilizadas para gerenciamento de memória em sistemas reais. Com o advento dos processadores multi-núcleo, computadores multiprocessados poderão se tornar a regra nos próximos anos, enquanto hoje são a exceção. A combinação desses fatores indica que aproveitar a capacidade destes sistemas multiprocessados para melhorar a eficiência do gerenciamento automático da memória é um caminho promissor. Entretanto, para que isso seja realizado, é necessário garantir que a sincronização requerida pelo sistema de gerenciamento da memória seja o mais eficiente possível, tanto para evitar que os ganhos possíveis de desempenho sejam perdidos, quanto para manter os sistemas adaptáveis para um crescente número de processadores.

Nesta dissertação foi apresentada uma arquitetura para coleta de referências em sistemas multiprocessados que utiliza operações eficientes de sincronização. O trabalho seguiu ao longo de cinco capítulos principais.

Após introduzir o objeto de estudo no Capítulo 1, o Capítulo 2 tratou do gerenciamento da memória em caráter mais geral, apresentando as alternativas existentes e as três técnicas

mais fundamentais para o gerenciamento automático, sendo uma dessas três a contagem de referências. As análises comparativas de vantagens e desvantagens entre as três técnicas revelaram uma deficiência fundamental do algoritmo de contagem de referências: a inabilidade de recuperar ciclos de objetos inativos que se referenciam mutuamente.

Este problema é resolvido utilizando-se os algoritmos para contagem de referências cíclicas, como mostrado no Capítulo 3. O algoritmo mostrado neste capítulo é o mais recente e mais eficiente de uma série de variações sobre a idéia principal, que é realizar varreduras locais em pontos do grafo da memória onde podem existir ciclos de objetos inativos. O capítulo concluiu com dois exemplos do algoritmo em funcionamento, para ilustrar seu mecanismo.

O Capítulo 4 tratou da extensão do algoritmo de contagem de referências cíclicas para sistemas multiprocessados, indicando inicialmente as diferenças entre coleta de lixo paralela e concorrente e as dificuldades encontradas para levar o algoritmo de contagem de referências para um contexto multiprocessado. Foi observado que a necessidade de sincronização pode inviabilizar o uso real de um algoritmo concorrente para gerenciamento da memória. Em seguida foi apresentada uma arquitetura que implementa a contagem de referências cíclicas em sistemas multiprocessados utilizando operações de sincronização mais eficientes do que semáforos, e que estão disponíveis nativamente em praticamente todos os processadores atuais. Embora a versão inicial da arquitetura seja mais adequada para computadores com apenas dois processadores, foram apresentadas variações que podem aproveitar com sucesso os recursos de um número maior de processadores.

Os testes realizados indicam consistentemente que a versão concorrente do algoritmo é mais eficiente que a versão seqüencial, em um computador com dois processadores. Isto indica que a sincronização utilizada é suficientemente eficiente para compensar o uso de um coletor de lixo concorrente baseado em contagem de referências cíclicas. Desta forma, conclui-se que a arquitetura proposta é válida e tem potencial de uso em situações reais.

6.2 Trabalhos Futuros

O estudo dos coletores de lixo paralelos e concorrentes, embora já antigo, tem se tornado mais importante nos últimos anos, o que se reflete em um maior número de artigos publicados e mais grupos de pesquisa voltados para a área. Se as tendências atuais da indústria de microprocessadores se mantiverem, os próximos anos verão uma mudança profunda nos computadores utilizados em ambientes não-especializados: enquanto hoje a grande maioria

dos computadores que não são dedicados a aplicações especializadas são seqüenciais e uniprocessados, no futuro todos se tornariam multiprocessados, através do uso dos processadores multi-núcleo. Isso cria uma necessidade urgente de adaptar o *software* para que acompanhe a evolução do *hardware* e, como comentado aqui, tornar a coleta de lixo paralela ou concorrente é uma maneira direta de aproveitar esse potencial, ao menos em parte; mais importante é que isto não requer nenhuma alteração nos programas já existentes, facilitando a transição.

Sendo uma área promissora, há muito o que estudar ainda. Algumas sugestões apresentadas aqui, das muitas que podem ser imaginadas, são:

- ▷ Comparar a arquitetura proposta nesta dissertação com outros algoritmos de coleta de lixo concorrente e paralela, incluindo os algoritmos de Lins [27, 29, 31].
- ▷ Estudar a formalização da arquitetura proposta, e provar sua corretude em um sistema formal adequado.
- ▷ Implementar a versão da arquitetura que utiliza vários mutadores para um coletor, observando até que número de mutadores é apropriado usar apenas um coletor.
- ▷ Projetar os detalhes de uma versão da arquitetura para coleta concorrente e paralela, utilizando vários coletores simultaneamente. Implementar esta versão e determinar empiricamente alguns parâmetros, como a razão entre o número de mutadores e o de coletores.

Referências

[1] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, USA, Abril 1967, pp. 483–485.

[2] G. R. Andrews, *Concurrent Programming: Principles and Practice.* Addison-Wesley Professional, Julho 1991.

[3] ——, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley Professional, Julho 1999.

[4] K. Arnold, J. Gosling, and D. Holmes, *Java(TM) Programming Language, The (4th Edition) (Java Series).* Addison-Wesley Professional, Agosto 2005.

[5] ArsTechnica, "Multicore, dual-core, and the future of intel," 2004. [Online]. Available: http://arstechnica.com/articles/paedia/cpu/intel-future.ars/1

[6] N. Benton, "Abstracting allocation: The new new thing," in *Computer Science Logic (CSL 2006)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2006.

[7] S. Blazy and X. Leroy, "Formal verification of a memory model for C-like imperative languages," in *International Conference on Formal Engineering Methods (ICFEM 2005)*, ser. Lecture Notes in Computer Science, vol. 3785. Springer-Verlag, 2005, pp. 280–299.

[8] H.-J. Boehm, "Destructors, finalizers, and synchronization," in *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 38, no. 1. New York, NY, USA: ACM Press, Janeiro 2003, pp. 262–272.

[9] ——, "The space cost of lazy reference counting," in *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 39, no. 1. New York, NY, USA: ACM Press, Janeiro 2004, pp. 210–219.

[10] G. E. Collins, "A method for overlapping and erasure of lists," *Communications of the ACM*, vol. 3, no. 12, pp. 655–657, Dezembro 1960.

[11] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, "Lock-free reference counting," *Distributed Computing*, vol. 15, no. 4, pp. 255–271, Dezembro 2002.

[12] J. DeTreville, "Experience with concurrent garbage collectors for Modula-2+," DECSRC, Tech. Rep. 64, Agosto 1990.

[13] P. L. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Commun. ACM*, vol. 19, no. 9, pp. 522–526, Setembro 1976.

[14] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, pp. 966–975, Novembro 1978.

[15] J. A. S. Filho, "Algoritmos para contagem de referências cíclicas," Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Fevereiro 2002.

[16] C. Flood, D. Detlefs, N. Shavit, and C. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001.

[17] W. Gruener, "Intel aims for 32 cores by 2010," *TG Daily*, 2006. [Online]. Available: http://www.tgdaily.com/2006/07/10/intel_32_core_processor/

[18] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# Programming Language, The (2nd Edition) (Microsoft .Net Development Series)*. Addison-Wesley Professional, Junho 2006.

[19] M. P. Herlihy and J. E. B. Moss, "Lock-free garbage collection for multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 3, pp. 304–311, 1992. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=139204

[20] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, Novembro 2000.

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

[22] S. P. Jones and J. H. (editors), Tech. Rep.

[23] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001. [Online]. Available: citeseer.ist.psu.edu/article/leijen01parsec.html

[24] Y. Levanoni and E. Petrank, "An on-the-fly reference-counting garbage collector for java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, pp. 1–69, Janeiro 2006. [Online]. Available: http://dx.doi.org/10.1145/1111596.1111597

[25] R. D. Lins, "Lazy cyclic reference counting," *Journal of Universal Computer Science*, vol. 9, no. 8, pp. 813–828, Agosto 2003.

[26] ——, "Efficient lazy cyclic reference counting," *Inf. Process. Lett.*, to appear.

[27] ——, "A shared memory architecture for parallel cyclic reference counting," *Microprocessing and Microprogramming*, vol. 34, pp. 31–35, Setembro 1991.

[28] ——, "Cyclic reference counting with lazy mark-scan," *Inf. Process. Lett.*, vol. 44, no. 4, pp. 215–220, Dezembro 1992.

[29] ——, "A multi-processor shared memory architecture for parallel cyclic reference counting," *Microprocessing and Microprogramming*, vol. 35, pp. 563–568, 1992.

[30] ——, "An efficient algorithm for cyclic reference counting," *Inf. Process. Lett.*, vol. 83, no. 3, pp. 145–150, Agosto 2002.

[31] ——, "A new multi-processor architecture for parallel lazy cyclic reference counting," in *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 35–43.

[32] A. D. Martínez, R. Wachenchauzer, and R. D. Lins, "Cyclic reference counting with local mark-scan," *Inf. Process. Lett.*, vol. 34, no. 1, pp. 31–35, Fevereiro 1990.

[33] Y. Matsumoto, *Ruby In A Nutshell*. O'Reilly Media, Inc., Novembro 2001.

[34] H. J. Mcbeth, "Letters to the editor: on the reference counter method," *Commun. ACM*, vol. 6, no. 9, Setembro 1963.

[35] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Communications of the ACM*, vol. 3, no. 3, pp. 184–195, Março 1960.

[36] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dezembro 1978.

[37] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=658762

[38] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 1996, pp. 2–11.

[39] W. Partain, "The nofib benchmark suite of haskell programs," in *Workshops in Computing – Proceedings of the 1992 Glasgow Workshop on Functional Programming*. London, UK: Springer Verlag, 1993, pp. 195–202.

[40] D. A. Patterson and J. L. Hennessy, *Organização e Projeto de Computadores: A Interface Hardware/Software, 3 ed.* Editora Campus, 2005.

[41] S. L. Peyton-Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Series)*. Prentice Hall, Maio 1987.

[42] G. L. Steele, "Multiprocessing compactifying garbage collection," *Commun. ACM*, vol. 18, no. 9, pp. 495–508, Setembro 1975.

[43] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, Setembro 2005. [Online]. Available: http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=332

[44] S. Thompson, *Haskell: The Craft of Functional Programming, 2 ed.* Addison-Wesley, Março 1999.

[45] J. D. Valois, "Implementing lock-free queues," in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, Outubro 1994, pp. 64–69. [Online]. Available: citeseer.ist.psu.edu/valois94implementing.html

[46] G. Van Rossum, *The Python Language Reference Manual*. Network Theory Ltd., Setembro 2003.

[47] J. Weizenbaum, "Symmetric list processor," *Commun. ACM*, vol. 6, no. 9, pp. 524–536, Setembro 1963.

APÊNDICE A

Implementação da Plataforma de Testes

Para realizar os testes da arquitetura descrita no Capítulo 4, foi implementado um compilador para uma linguagem funcional *lazy* simples, para que fosse possível ter todo o controle sobre a implementação do coletor de lixo de maneira concorrente. Vários compiladores e sistemas de tempo de execução atuais não são preparados para aproveitarem múltiplos processadores, se presentes. Isto motivou a criação de uma implementação completa, do compilador ao sistema de tempo de execução, para aproveitar os sistemas multiprocessados.

A linguagem implementada segue o paradigma funcional e é similar a uma versão simplificada da linguagem Haskell [22, 44]. O compilador analisa a sintaxe e verifica os tipos do programa, depois realiza o processo de *lambda-lifting* nas funções do programa, traduzindo o resultado para código da máquina G; este código é então traduzido para código nativo da arquitetura Intel de 32 bits. O resultado final é ligado com o sistema de tempo de execução e o coletor de lixo, que foram escritos em C. As técnicas de compilação utilizadas são conhecidas e podem ser encontradas, por exemplo, no livro de Peyton-Jones [41].

Para a implementação das *threads* concorrentes, foi utilizada a biblioteca *POSIX Threads*, cujas versões mais recentes no sistema operacional Linux aproveitam e utilizam vários processadores, se presentes no computador. Com isto, fica garantido que *threads* diferentes podem ser agendadas para processadores diferentes, dependendo da disponibilidade dos mesmos.

As seções seguintes descrevem a implementação em maiores detalhes.

Módulo	Descrição
AbsSyntax	Definições para a representação em sintaxe abstrata dos programas de entrada
Compiler	Módulo principal do compilador
ELambda	Definições e funções para representação do programa como uma expressão em um λ-*calculus* extendido
Environ	Funções para usar e manipular ambientes (*environments*)
GCode	Tradução do programa para o código de entrada da máquina G (*código G*)
GPrims	Funções primitivas da linguagem implementadas em código G
Parser	Analisador sintático
Primitives	Descrições de alto nível das funções primitivas da linguagem
SuperComb	*Lambda-lifting* e tradução do programa para super-combinadores
Type	Verificação e reconstrução de tipos, seguindo o sistema de tipos Hindley-Milner
Util	Funções utilitárias usadas em vários outros módulos
X86	Gerador de código para a arquitetura Intel x86

Tabela A.1: *Módulos do compilador e suas descrições*

A.1 Sintaxe

A Figura A.1 mostra a sintaxe da linguagem de teste utilizada, que é a linguagem de entrada do compilador implementado. Pode-se perceber que a sintaxe é similar à da linguagem Haskell; entretanto, decidiu-se usar ponto-e-vírgula como terminador de linhas obrigatório, para evitar a implementação de uma regra de *layout* complexa (vide definição da linguagem Haskell [22]).

A.2 Compilador

O compilador para a linguagem de teste foi escrito em Haskell, linguagem que se mostrou bastante adequada para a tarefa: o compilador completo tem pouco mais de duas mil linhas de código, incluindo o analisador sintático. Ao total, o compilador é composto por 12 módulos, como mostrado na Tabela A.1.

O módulo principal, *Compiler*, contém a interface de linha de comando e realiza a coordenação entre os outros módulos para gerar o código final. As opções na linha de comando determinam que etapas da compilação serão realizadas, ou seja, que estágios do compilador serão utilizados. Por exemplo, é possível fazer com que o compilador apenas faça a análise sintática, ou apenas a verificação e reconstrução de tipos, ou chegue até a geração do código G, e assim em diante. A Figura A.2 mostra a função principal do compilador; seu funcionamento

```
# Syntaxe (em EBNF)

# id e typevar são identificadores de variáveis, começando com uma letra minúscula
# typeid é um identificador de construtor de tipo
# consid é um identificador de construtor de dados
# tanto typeid quanto consid devem começar com uma letra maiúscula

<program> := <decl>*

<decl> := [<typeattr> ;] <valdecl> ;    # declaração de um valor
        | <typedecl> ;                  # declaração de um tipo

<typeattr>  := id :: <type>

<type> := typeid <type>*                        # tipos construídos
       | [<type>]                               # tipos de lista
       | <type> -> <type>                       # tipos de função
       | (<type>{,<type>}*)                     # tipos produto (tuplas)
       | typevar | Int | Bool

<typedecl> := data typeid typevar* = <variant> { | <variant> }*
            | type typeid typevar* = <type> | typevar

<variant> := consid <type>*

<valdecl> := id <pat>* = <exp>

<pat> := id

<exp> := id
       | <integer> | <boolean> | <exp> + <exp> | <exp> * <exp> | <exp> - <exp>
       | <exp> / <exp> | - <exp> | <exp> && <exp> | <exp> || <exp> | not <exp>
       | <exp> = <exp> | <exp> /= <exp> | <exp> > <exp> | <exp> < <exp>
       | <exp> >= <exp> | <exp> <= <exp> | (<exp>)
       | consid <exp>*                  # construtor arbitrário
       | <exp> : <exp>                  # construção de lista
       | if <exp> then <exp> else <exp> # condicional
       | let <pat> = <exp> in <exp>     # expressão qualificada
       | let id <pat>* = <exp> in <exp>
       | <exp> <exp>* # aplicação
       | (<exp>{,<exp>}+)               # tupla
       | [<exp>{,<exp>}*]               # lista

<boolean>   := True
             | False
```

Figura A.1: *Sintaxe para a linguagem de entrada do compilador em notação BNF*

consiste em obter os argumentos e opções da linha de comando e determinar que estágio do compilador deve ser chamado, em seguida chamando a função `compile` para executar o estágio selecionado.

```
main = do args <- getArgs
          prog <- getProgName
          (o, n) <- getOptions args prog
          (stage, o') <- compileStage o
          if null n then putStrLn (usage prog)
           else compile stage o' (head n) prog
```

Figura A.2: *Função principal do compilador*

A função `compile` está mostrada na Figura A.3, que consiste em realizar a análise sintática no programa (através da função `parse`), seguida pela verificação e reconstrução de tipos (função `typeCheck`) e, ao final, chamando a função `dispatch` para encaminhar o programa para o estágio selecionado do compilador.

```
compile :: CompilerStage -> [Flag] -> String -> String -> IO ()
compile stage opts file prog = do e <- parse file
                                  t <- typeCheck e opts
                                  o <- getOutput opts
                                  dispatch stage e t o
                                  hClose o
```

Figura A.3: *Função `compile`*

O analisador sintático foi construído utilizando a biblioteca *Parsec*, que implementa combinadores monádicos de analisadores sintáticos [23]. A verificação e reconstrução de tipos utiliza o algoritmo de reconstrução de Milner [36].

A função `dispatch` chama o estágio adequado do compilador, segundo as opções utilizadas pelo usuário. A definição da função `dispatch` é mostrada na Figura A.4, juntamente com algumas funções auxiliares. Como o objetivo final do compilador é gerar código *assembly* para que seja compilado para linguagem de máquina, é este estágio final que será descrito.

A entrada da função `dispatch` é composta pela representação do programa em sintaxe abstrata (variável `e`), o tipo determinado para o programa (variável `t`) e um fluxo de saída, que pode ser um arquivo ou o console (variável `out`). O último caso para a função `dispatch` trata o estágio `Assembly`, cujo objetivo é gerar código *assembly* para o programa. Vê-se facilmente

```
intForm e = ELambda.translateProg e
superclift i = SuperComb.translate i
supercomb i = SuperComb.translate i
gcode s = GCode.translate s
asm gi = X86.translate (GPrims.primitiveCode ++ gi)

dispatch TypeAnalysis e t out = hPutStrLn out (showType t)
dispatch IntermForm e t out = hPutStrLn out (ELambda.prettyPrint $ intForm e)
dispatch SuperComb e t out = let sc = superclift $ intForm e in
                               hPutStrLn out (SuperComb.printSCProg sc)
dispatch G_Code e t out = let is = gcode $ supercomb $ intForm e in
                          hPutStrLn out (GCode.printInstructions is)
dispatch Assembly e t out = let a = asm $ gcode $ supercomb $ intForm e in
                            hPutStrLn out (X86.printInstructions a)
```

Figura A.4: *Função* `dispatch`

que o seguinte processo é seguido nesse caso:

1. A representação em sintaxe abstrata é traduzida para uma forma intermediária baseada em um λ-*calculus* extendido;
2. Esta forma intermediária passa pelo processo de *lambda-lifting*, gerando um conjunto de super-combinadores;
3. Os super-combinadores resultantes são traduzidos para código G, o código de entrada da máquina G;
4. O código G é traduzido para código *assembly* para processadores Intel da arquitetura x86.

O código *assembly* final é compatível com o programa AS, o *assembler* do projeto GNU.

A.3 Suporte de tempo de execução

O suporte de tempo de execução para os programas compilados consiste basicamente de duas partes:

1. A máquina de redução de grafos
2. O coletor de lixo

A máquina de redução de grafos é uma implementação da máquina G, e suas operações estão incluídas no código *assembly* gerado ao final da compilação. O coletor de lixo, por sua vez, foi escrito em linguagem C de forma a cooperar com o programa assembly gerado pelo compilador.

A.3.1 Formato das células

Os objetos manipulados pela máquina G são células de tamanho fixo. O coletor de lixo utiliza esse fato para realizar o gerenciamento da memória, como foi mencionado nos Capítulos 2 e 3. A Figura A.5 mostra graficamente a estrutura dessas células, e a Figura A.6 mostra a mesma estrutura como expressa no programa em linguagem C. A célula é formada por quatro campos de 32 bits cada:

- O campo *gc* é reservado para o uso do coletor de lixo
- O campo *tag* é utilizado como um identificador do tipo do objeto que ocupa a célula
- Os campos *field1* e *field2* têm significados diferentes dependendo do tipo do objeto que ocupa a célula, ou seja, as funções destes campos dependem do valor do campo *tag*

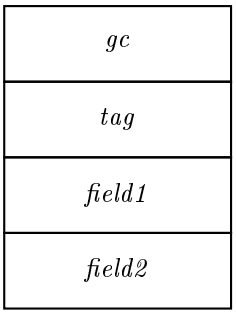


Figura A.5: *Formato das células*

```
typedef struct tagCell
{
  unsigned gc;
  unsigned tag;
  void *   field1;
  void *   field2;
} Cell;
```

Figura A.6: *Declaração da estrutura das células em C*

A reserva de um campo de 32 bits para o coletor de lixo permite integrar ao código gerado pelo compilador coletores utilizando qualquer uma das técnicas mostradas no Capítulo 2. O campo *tag* é utilizado não só para identificar o tipo do objeto, como também para apontar para uma *tabela de despacho*, que contém os endereços de versões específicas para o tipo em questão de rotinas genéricas na linguagem. Este uso é similar à implementação original da máquina G – vide Capítulo 19 do livro de Peyton-Jones [41]. Os tipos disponíveis na implementação – cada um possui uma *tag* correspondente – são:

▷ inteiro

▷ valor *booleano*

▷ função

▷ aplicação

▷ construção de lista (célula *cons*)

A.3.2 Organização

O suporte de tempo de execução realiza a inicialização do programa como um todo, chamando a função de inicialização do coletor de lixo e iniciando a execução do código gerado pelo compilador. A função principal do sistema de suporte é mostrada na Figura A.7. As funções `Initialize` e `CleanUp` são parte do coletor de lixo, enquanto que a função `faul_main` é declarada pelo código gerado pelo compilador. Além da função principal, o suporte de execução inclui algumas funções auxiliares necessárias durante a execução do programa, como algumas rotinas de impressão simples.

```
int main(void)
{
  Initialize();

  faul_main(stack_end);

  CleanUp();
}
```

Figura A.7: *Função principal do suporte de tempo de execução*

A.3.3 Coletor de lixo

Outra função do suporte de tempo de execução é o gerenciamento da memória, que é realizado pelo coletor de lixo. Para possibilitar que vários coletores diferentes pudessem ser integrados ao programa, foi definida uma interface simples que deve ser implementada por qualquer coletor. Esta interface é mostrada na Figura A.8.

```
int main(void)
void Initialize(void);
Cell * New();
void Update(void *ptr, Cell *c);
void Delete(Cell *c);
void CleanUp(void);
```

Figura A.8: *Interface para o coletor de lixo*

Foram implementadas quatro versões do coletor de lixo:

1. Um coletor que não faz nenhuma tarefa de gerenciamento da memória; na verdade, isto significa que não há coletor de lixo
2. Um coletor baseado na contagem de referências simples, como mostrada no Capítulo 2; este coletor não recupera ciclos de células inativas
3. Um coletor baseado na contagem de referências cíclicas, seguindo a especificação mostrada no Capítulo 3
4. Um coletor concorrente que executa em um processo separado do único processo mutador, correspondendo à versão da arquitetura com um coletor e um mutador do Capítulo 4

A função `Initialize` para o coletor concorrente cria uma nova *thread* para o coletor chamando a função `pthread_create` da biblioteca *POSIX Threads* ou *pthreads*. As pilhas de incremento e decremento são implementadas como listas duplamente encadeadas, com mutador e coletor tendo acesso a pontas opostas de cada uma, como explicado no Capítulo 4. Embora não seja necessário utilizar sincronização explícita nessa arquitetura para as operações do coletor, a Figura A.9 mostra a implementação de uma operação atômica do tipo *compare-and-swap* para a arquitura Intel x86. Esta função pode ser extendida para trabalhar com outras arquiteturas sem maiores dificuldades, já que todas as arquiteturas atuais possuem alguma operação do tipo *compare-and-swap* no seu conjunto de instruções.

```
bool_t CompareAndSwap(IN void ** ptr, IN void * old, IN void * new)
{
   unsigned char ret;

   __asm__ __volatile__
   (
      "  lock\n"
      "  cmpxchgl %2,%1\n"
      "  sete %0\n"
      : "=q" (ret), "=m" (*ptr)
      : "r" (new), "m" (*ptr), "a" (old)
      : "memory"
   );

   return ret;
}
```

Figura A.9: *Implementação da operação* compare-and-swap

APÊNDICE B

PROGRAMAS DE TESTE

Aqui são apresentados e descritos os programas utilizados para testar o desempenho da arquitetura proposta, cujos resultados são mostrados no Capítulo 5. Os programas foram escritos na linguagem descrita no Apêndice A.

B.1 Função de Ackermann

Este programa implementa a *função de Ackermann*, um exemplo de função computável que não apresenta recursão primitiva. A função recebe dois números naturais como entrada e produz outro número natural. O texto do programa é mostrado na Figura B.1.

```
-- Funcao de Ackermann: acker
acker m n = if m == 0 then (n + 1) else
            if m > 0 && n == 0 then acker (m - 1) 1 else
            acker (m - 1) (acker m (n - 1));

main = acker 3 4;
```

Figura B.1: *Programa* acker

Este programa é interessante pois gera muitas chamadas recursivas, o que testa a capacidade de recuperar ciclos. Por outro lado, o número de chamadas cresce muito rapidamente, o que provoca um estouro na pilha do programa para valores de m tão pequenos quanto 5. Muitas implementações de linguagens de programação, quando usam uma pilha de tamanho padrão, também encontram erros ao tentar calcular valores maiores desta função.

B.2 Concatenação de listas

O programa na Figura B.2 chama repetidamente uma função para concatenar listas, aplicando várias funções sobre seus componentes. Ao utilizar recursividade e listas, geram-se várias referências a células que devem ser gerenciadas pelo coletor de lixo.

```
-- conctwice
conc l1 l2 = if null l1 then l2 else (head l1) : (conc (tail l1) l2);
map f l = if null l then [] else (f (head l)) : (map f (tail l));
twice f x = f (f x);
sq x = x * x;
succ x = x + 1;
pred x = x - 1;
fib n = if n < 2 then 1 else (fib (n - 1)) + (fib (n - 2));
main = conc (conc (conc (map fib [10, 11, 12, 13, 14]) (map sq [4, 5, 6, 7]))
                  (map (twice succ) [2, 3, 4, 5]))
    (map (twice pred) [2, 3, 4, 5]);
```

Figura B.2: *Programa* conctwice

B.3 Números de Fibonacci

Este programa calcula vários números da seqüência de Fibonacci, utilizando uma função recursiva. As duas chamadas recursivas na função `fib` tornam sua complexidade exponencial, gerando um grande número de ciclos no grafo da memória, e exigindo bastante do coletor de lixo. O programa é mostrado na Figura B.3.

```
-- fiblista
map f l = if null l then [] else (f (head l)) : (map f (tail l));
fib n = if n < 2 then 1 else (fib (n - 1)) + (fib (n - 2));
main = map fib [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26];
```

Figura B.3: *Programa* fiblista

B.4 Cálculo do fatorial

Este programa cria um padrão de chamadas recursivas similar ao do programa para cálculo do número de Fibonacci, mas usando o cálculo do fatorial. Neste caso, os números crescem

muito mais rapidamente, causando um estouro da capacidade de palavras de 32 bits. Por esta razão o programa apenas repete o cálculo para um mesmo argumento, como mostrado na Figura B.4.

```
-- recfat
map f l = if null l then [] else (f (head l)) : (map f (tail l));
fat n = if n < 2 then 1 else (n * (fat (n - 1)));
recfat n = if n <= 0 then (fat 0) else (fat n) + (recfat (n - 1)) + (recfat (n - 2));
run f x n = if n == 0 then [] else (f x) : (run f x (n - 1));
main = run recfat 12 12;
```

Figura B.4: *Programa* recfat

B.5 Somatório recursivo

Um programa para calcular recursivamente o somatório de números variando de m até n, com incrementos de 1. A Figura B.5 lista todo o texto do programa, que aplica a função de somatório a vários pares de números.

```
-- somatorio
map f l = if null l then [] else (f (head l)) : (map f (tail l));
somatorio m n = if n == m then n else (m + (somatorio (m + 1) n));
main = map (somatorio 1) [150, 160, 170, 180, 190, 200, 210, 220, 230, 240];
```

Figura B.5: *Programa* somatorio

B.6 Somatório de listas

Este programa calcula os somatórios S_n de todos os números entre 1 e n, para sucessivos valores de n. O texto se encontra na Figura B.6.

```
-- somamap
map f l = if null l then [] else (f (head l)) : (map f (tail l));
gl l x = if x == 0 then l else (gl (x : l) (x - 1));
somatorio m n = if n == m then n else (m + (somatorio (m + 1) n));
main = map (somatorio 1) (gl [] 100);
```

Figura B.6: *Programa* somamap

B.7 Programa *tak*

Este programa foi retirado do conjunto de *benchmarks* **nofib**, criado por Partain [39] para medir o desempenho de implementações de linguagens funcionais *lazy*. O programa não calcula nada em específico, mas gera um número de chamadas recursivas grande, mas menor que o da função de Ackermann, resultando na possibilidade de execução com argumentos maiores.

```
tak x y z = if not(y < x) then z
            else tak (tak (x-1) y z)
                     (tak (y-1) z x)
                     (tak (z-1) x y);

main = tak 17 16 8;
```

Figura B.7: *Programa* tak

B.8 Problema das N rainhas

O programa que calcula o número de soluções para o problema das N rainhas também foi inspirado em um programa similar incluído no conjunto de *benchmarks* **nofib**, mas precisou de muitas adaptações pois o original usa muitas características da linguagem Haskell que não estão disponíveis na linguagem dos testes.

```
map f l = if null l then [] else (f (head l)) : (map f (tail l));

filter p l = if null l then []
             else let lh = head l in
                  let lt = tail l in
                  if (p lh) then (lh : (filter p lt)) else (filter p lt);

range n = let aux i = if i > n then [] else (i : (aux (i+1))) in aux 1;

safe d l = if null l then True
             else let x  = head l in
                  let lh = head (tail l) in
                  let lt = tail (tail l) in
                  x /= lh && x /= lh+d && x /= lh-d && (safe x (d+1) lt);

safe1 l = safe 1 l;

cons q l = q : l;

gen n nq = if n == 0 then [[]]
           else let f l = (let g x = (x : l) in map g (range nq)) in
           filter safe1 (map f (gen (n-1) nq));

main = len (gen 10 10);
```

Figura B.8: *Programa* queens

Sobre o Autor

O autor nasceu em João Pessoa, Paraíba, no dia 25 de agosto de 1977. Formado em Engenharia Elétrica com habilitação em Controle e Automação pela Universidade Federal de Campina Grande (UFPE).

Entre suas áreas de interesse estão a teoria e implementação das linguagens de programação, lógica, teoria dos tipos, semântica, métodos formais e sistemas distribuídos.

Endereço: Rua João Francisco Lisboa, 121 bl. 16 ap. 02
Várzea
Recife – PE, Brasil
C.E.P.: 50.741 – 100

e-mail: andrei.formiga@gmail.com

Esta dissertação foi diagramada usando LaTeX 2_ε[1] pelo autor.

[1] LaTeX 2_ε é uma extensão do LaTeX. LaTeX é uma coleção de macros criadas por Leslie Lamport para o sistema TeX, que foi desenvolvido por Donald E. Knuth. TeX é uma marca registrada da Sociedade Americana de Matemática ($\mathcal{AMS}$). O estilo usado na formatação desta dissertação foi escrito por Dinesh Das, Universidade do Texas. Modificado em 2001 por Renato José de Sobral Cintra, Universidade Federal de Pernambuco, e em 2005 por André Leite Wanderley.