
Pós-Graduação em Ciência da Computação

**Filipe Rafael Gomes Varjão**

# Gerenciamento dinâmico de memória baseado em regiões com contagem de referências cíclicas

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

**Filipe Rafael Gomes Varjão**

# Gerenciamento dinâmico de memória baseado em regiões com contagem de referências cíclicas

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

**Área de Concentração**: Engenharia de Software
**Orientador**: Prof. Dr. Rafael Dueire Lins

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

V313g Varjão, Filipe Rafael Gomes
Gerenciamento dinâmico de memória baseado em regiões com contagem de referências cíclicas / Filipe Rafael Gomes Varjão. – 2019.
108 f.: il., fig., tab.

Orientador: Rafael Dueire Lins.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências e apêndices.

1. Engenharia de software. 2. Gerenciamento de memória. I. Lins, Rafael Dueire (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-097

**Filipe Rafael Gomes Varjão**

**“Gerenciamento Dinâmico de Memória Baseado em Regiões com Contagem de Referências Cíclicas”**

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 14/03/2019.

_____

**Orientador: Prof. Rafael Dueire Lins**

**BANCA EXAMINADORA**

_____

Prof. Dr. André Luis de Medeiros Santos
Centro de Informática /UFPE

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE

_____

Prof. Dr. Francisco Heron de Carvalho Junior
Departamento de Computação / UFC

_____

Prof. Dr. Siang Wun Song
Instituto de Matemática e Estatística / USP

_____

Prof. Dr. Fabiano Amorim Vaz
Instituto Federal da Bahia / Campus Paulo Afonso

*Dedico este trabalho à minha família, esposa e filhas, que foram porto seguro perante as dificuldades durante este percurso.*

## AGRADECIMENTOS

Primeiramente, agradeço a DEUS pelo dom da vida, com sua misericórdia infinita me sustenta e tem derramado graças abundantes sobre todos nós.

A meu pai e mãe pelo constante incentivo e suporte para seguir em frente na vida e nos estudos. Juntos, sempre acreditaram em mim: meu irmão Thiago e sua linda família, Juliana, Arthur, Victor, Samuel e Bento. Minha irmã Fabiana que, mesmo distante, sempre presente em minha vida, assim como Luciano seu esposo e meu sobrinho Ítalo.

Gostaria encarecidamente de agradecer ao Professor Rafael Dueire Lins, cuja dedicação e empenho tornaram possível a conclusão dessa árdua jornada.

Também gostaria de dizer obrigado a muitos envolvidos diretamente com esse trabalho sempre dispostos a me ajudar e tirar dúvidas, Fabiano, Camila Araujo, Daker Fernandes, João Leonardo (Leu) e aos desenvolvedores do Google, em especial a Austin Clements, Cherry Zhang e Keith Randall. Sou grato a todos a que conheci e com os quais pude conviver nesta etapa da minha vida no laboratório do Centro de Informática: Bruno Maciel, Sidartha de Carvalho, Wanessa, Paulo Henrique, Ruan, Renê, Hilário. Assim como muitos outros amigos próximos que sempre estiveram em oração por minha alma, Dona Ana minha santa sogra, Aline Andrade Alves, José Carlos, Cícero, Samuel Almeida, Celivane e muitos outros.

De forma especial agradecer a Suzana, minha esposa, por toda a paciência, amor e carinho; e a minhas princesas Letícia, Sara e Helena, que me fazem querer ser melhor a cada dia.

A todos meu sincero muito obrigado, pois, sozinho, há muito teria desistido, principalmente pela amizade e carinho nunca a mim negado.

*Eu acredito que às vezes, são as pessoas de quem ninguém espera nada que fazem as coisas que ninguém consegue imaginar (HODGES, 1983).*

## RESUMO

Desde o algoritmo de Mark-Scan desenvolvido para a linguagem LISP por John McMarthy em 1960, diversas técnicas foram desenvolvidas para gerenciar dinamicamente a memória de forma automática. Tais técnicas buscam coletar todos os dados, mantendo as pausas do programa de usuário mínimo. A contagem de referência é um método simples, no qual o gerenciamento de memória é realizado em pequenas etapas intercaladas com computação. Esse método evita grandes pausas de programa introduzidas por outras técnicas, como a marcação e varredura e a coleta por cópia. No entanto, a versão original da contagem de referência é incapaz de recuperar estruturas cíclicas. A solução geral para tal problema foi a introdução da varredura local. Combinando a contagem de referência com detecção cíclica e o gerenciamento de memória baseado em região, é possível diminuir o número de verificações e atualizações nas contagens das células em *heap*, reduzindo a duração das pausas e o espaço em memória. O gerenciamento de memória baseado em região faz uso de análise estática e instrumentação de código. O compilador tenta identificar a vida útil dos objetos, agrupando aqueles de longevidade semelhante em uma mesma região (geralmente contígua) de memória a ser recuperada quando não forem mais necessárias. Esta tese propõe um novo algoritmo de gerenciamento de memória baseado na contagem de referências cíclicas, combinado com o gerenciamento de memória baseado em região, fazendo uso de técnicas estáticas e de tempo de execução, trabalhando em conjunto para reduzir os gargalos um do outro. A mesclagem de contagem de referência e gerenciamento de memória baseado em região foi usada para coletar os dados alocados em grupos. As duas técnicas são usadas para reciclar as diferentes partes da memória, aumentando a velocidade de recuperação de objetos e reduzindo o uso da memória. Com o compilador implementado em Go, são inseridas barreiras de gravação após cada operação de armazenamento de um ponteiro na memória, atualizando as contagens de referência de todas as células envolvidas. O alocador de células segrega os objetos por seu tamanho, gerenciando listas que contêm os blocos livres de um tamanho específico com a contagem de referência armazenada em cada objeto. Essa modificação limita a varredura de marca local apenas a ciclos de pesquisa preguiçosos, evitando varrer todo o *heap*. Em comparação com o algoritmo de contagem de referências e detecção de ciclos, observam-se ganhos de desempenho em até 30% em tempo de execução e 50% a menos de uso de memória para programas que exigem grandes quantidades de memória e não fazem pausas no processo do usuário, esses valores podem apresentar maiores ganhos quando são combinadas as técnicas de contagem de referências e o gerenciamento de memória baseado em regiões.



**Palavras-chaves**: Coleta de lixo. Gerenciamento de memória. Contagem de referência.

## ABSTRACT

From the Mark-Scan algorithm developed for the LISP language by John McMarthy in 1960, many techniques have been developed to proper and automatically manage memory. Such techniques seek to collect all possible unreachable data, keeping program pauses to a minimum. Reference counting is a simple method, in which memory management is performed in small steps interleaved with computation, it avoids the large program pauses introduced by other techniques such as mark-sweep and copying. However, the original version of reference counting is unable to reclaim cyclic structures. The general solution for such a problem was introducing local mark-sweep. Combining the reference counting with cyclic detection and the region-based memory management, which decreases the number of checks and updates to the counts of the cells in heap, reducing the length of pauses and memory footprint. Region-based memory management makes use of static analysis and runtime instrumentation. The compiler tries to identify the lifetime of objects, clustering the ones with similar life span in the same region (often contiguous) of memory to be reclaimed when they are no longer needed. The merge of reference counting and region-based memory management was used in to collect the data allocated in groups. Here, both techniques are used to recycle the different parts of the memory, increasing the speed of object reclamation while also decreasing the overall footprint of the program. Region-based memory management, a compile-time technique, and reference counting, a run-time algorithm, work together to eliminate some of the disadvantages of each other. In the implementation reported here, the Go compiler inserts a write barrier after every store operation of a pointer into memory, updating the reference counts of all the cells involved. The cell allocator also segregates the objects by their size, managing lists which hold the free blocks of a particular size with the reference count stored in each object. This modification limits the local mark-sweep to lazily search cycles only, avoiding to scan the entire heap. This thesis reports on a new memory management algorithm based on cyclic reference counting combined with region-based memory management, which makes use of static and runtime techniques, working together to reduce the bottlenecks of one another. The presented scheme yields performance gains for programs that claim for large quantities of memory as 30% off to runtime and 50% less of memory usage, those values can grow when is combined reference counting and region-based memory management, it makes no pauses of the user process, an essential feature for real-time systems.



**Keywords**: Garbage collection. Memory management. Reference counting.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| $GB$ | Gigabyte |
| $KB$ | Kilobyte |
| $MB$ | Megabyte |
| $ms$ | Milisegundos |
| **API** | Application Programming Interface |
| **CR** | Contagem de Referência |
| **CSP** | Communicating Sequential Processes |
| **GC** | Garbage Collection |
| **GCC** | GNU Compiler Collection |
| **Gccgo** | Go frontend |
| **IoT** | Internet of Things |
| **MS** | Mark-Sweep |
| **RAM** | Random Access Memory |
| **SO** | Sistema Operacional |

## SUMÁRIO

# 1 INTRODUÇÃO

O gerenciamento de memória é fundamental para todos os tipos de programas, porém, em sistemas concorrentes e de tempo real, o uso de coletores que precisa parar toda a computação para efetivar a limpeza da memória e pode caracterizar um sério problema em determinadas aplicações. O principal problema destacado consiste em pausas feitas para que o meio de gerência de memória, chamado de coletor, efetue a limpeza sem apresentar pausas que são indesejadas. Quando não for possível eliminar as pausas, é preciso reduzir a ocorrência ao mínimo.

O número de pessoas conectadas à Internet através de dispositivos demandando serviços distribuídos cada vez mais sofisticados é uma realidade hoje (MCLUHAN, 2019). O alto crescimento de múltiplos dispositivos também fez crescer o desenvolvimento e aprimoramento de *software* embarcado. Conjuntamente a esse crescimento, surge a necessidade de sistemas robustos, com maior poder de processamento, que suportem milhões de acessos simultâneos, ou em tempo real que rodam de maneira ininterrupta, ou seja, nunca finalizam suas execuções.

Landin (1966) já classificava diferentes famílias de linguagens em 1966, porém, hoje em dia, as linguagens têm aderido a diferentes abordagens e características. Dentre as linguagens, temos as orientadas a objetos, com funções *lambda*, função de alta ordem, diferentes algoritmos para coleta, métodos de sincronização entre *threads*, dentre outras.

Linguagens de programação apresentam constantes melhorias e aprimoramentos para apresentar suporte a alta concorrência, processamento em um ambiente distribuído ou com múltiplos núcleos, incorporando diversos recursos às linguagens para melhor gerir o consumo dos recursos disponíveis, principalmente em termos do uso do *multi-core* e gerenciamento automático de memória (TIOBE, 2019; REDMONK, 2018).

Um dos recursos fundamentais para a computação é o gerenciamento de memória, pois ele permite a continuidade da execução dos programas. O gerenciamento da memória é fundamental principalmente em sistemas de tempo real com alta taxa de disponibilidade. Prover um serviço de tempo real significa que, se um conjunto de tarefas for agendável, então as tarefas são escaláveis, respeitando as prioridades, prazos e condições de pior tempo de execução. Assim, o tempo de resposta é igualmente importante para um sistema robusto de tempo real.

Há duas possibilidades de gerenciamento de memória como recurso de um programa: o gerenciamento manual e automático. No gerenciamento manual o processo de alocamento e desalocamento da memória é feito pelo desenvolvedor. Essa abordagem requer experiência do programador e pode distanciar o foco do desenvolvimento da rotina principal, além disso, pode agregar maior complexidade estando fadado a erros. No gerenciamento dinâmico de memória, também chamado de coletor de lixo (Garbage Collection (GC)), é

realizada a coleta e reúso da memória de forma automática. Isso permite ao desenvolvedor manter a atenção nas rotinas do programa a ser desenvolvido, trazendo maior robustez e segurança à programação.

Não obstante, o uso do coletor de lixo é totalmente ligado à linguagem escolhida, ou seja, não são todas as linguagens de programação que dão suporte à coleta dinâmica de memória. Por exemplo, as linguagens como C e C++ não utilizam essa abordagem. Por outro lado, dentre as linguagens mais utilizadas atualmente temos Java, Python e Ruby, as quais apresentam suporte ao gerenciamento de memória.

Segundo Sammet (SAMMET, 1969) (apud (JONES; LINS, 1996, p.1)), "Uma das contribuições mais duradouras da linguagem de programação LISP é uma característica não linguística: o termo e a técnica de coleta de lixo, que se refere ao método do sistema de lidar automaticamente com o armazenamento". Linguagens com estruturas de dados dinâmicas, a partir de LISP e suas listas, usavam uma organização de memória chamada de *heap*, que era adequada para alocação e liberação de partes da memória em momentos arbitrários durante sua execução do programa.

Algumas técnicas de coleta de lixo têm sido estudadas desde então. A seguir alguns, exemplos:

- **Contagem de Referências (CR)**: Collins (1960), Martinez, Wachenchauzer e Lins (1990), Lins (1992a), Jones e Lins (1993), Shahriyar, Blackburn e Frampton (2013), Ugawa, Jones e Ritson (2014), Férey e Shankar (2016)
- **Marcação e varredura**: McCarthy (1960), Pizlo et al. (2010), Garner, Blackburn e Frampton (2011)
- **Cópia**: Minsky (1963), Fenichel e Yochelson (1969), Tahboub (1993), Ugawa, Iwasaki e Yuasa (2010)
- **Geracional**: Appel (1989), Clinger e Hansen (1997), Marlow et al. (2008), Ferreiro et al. (2016)

Essas técnicas compõem o estado da arte na área de gerenciamento de memória e norteiam o uso de coletas de memória dinâmica. Paradas *stop-the-world* que ocorrem durante a execução do programa, para que aconteça a coleta do lixo na memória, têm aberto estudos com abordagens híbridas de modo a reduzir (LANG; QUEINNEC; PIQUER, 1992; FLOOD et al., 2001) o tempo de parada dos programas para coleta (DOMANI et al., 2001; GIDRA et al., 2013).

Jones e Lins (1996) definem um coletor de lixo como o controle automático de armazenamento de memória dinamicamente. O GC tem reputação de ocupar uma grande carga de trabalho nos programas, assim, técnicas modernas de GC têm reduzido substancialmente o tempo de pausa do programa para a ação do coletor, porém muitos programas

escritos perdem desempenho devido a outros problemas como um mecanismo de passagem de parâmetros ineficiente, a falta de suporte a funções de alta ordem e a avaliação de expressões desnecessárias. Isso pode ser visto em linguagem de programação como: Python, Swift, dentre outras.

Um exemplo de linguagem de programação moderna com recursos de coleta automática é a linguagem Golang, desenvolvida por Pike (2009), no Google em 2009. Inicialmente denominada de Golang e também chamada por Go, a linguagem foi desenvolvida ainda em 2007, porém, somente foi lançada oficialmente em 2009 com licença *open source*. É uma linguagem procedural similar à linguagem C, que provê suporte a programação concorrente por meio de canais (*channels*), *interfaces*, criação de tipos e funções de alta ordem (PIKE, 2009; DONOVAN; KERNIGHAN, 2015). Além disso, Go é uma linguagem imperativa, estaticamente tipada, de propósitos gerais (*Web*, *desktop*, sistemas embarcados e outros), com uma comunidade de desenvolvedores ativa e com bibliotecas disponíveis de variados serviços.

O GC usado na linguagem Go é baseado no algoritmo de marcação e varredura, Mark-Sweep (MS), com o algoritmo *Tri-color* (DIJKSTRA et al., 1978), o algoritmo de marcação e varredura é simples, porém faz uso de pausas nos processos em execução. Essas pausas geralmente são indesejáveis, podendo levar a um tempo de pausa demasiadamente longo por percorrer a *heap* inteira.

Apesar da atual implementação do coletor em Go apresentar tempo de pausa inferior a 10 Milisegundos ($ms$) Sewell (2019), sistemas de tempo real devem corresponder a uma resposta rápida, dado o estímulo ou a entrada e a resposta ou a saída correspondente; logo, o tempo de resposta é um dos principais requerimentos em sistemas dessa natureza (JAFFE et al., 1991).

Pausas são indesejadas ou reduzidas à sua mínima ocorrência possível em sistemas como sistemas aeroespaciais, automotivos e ferroviários, sistemas espaciais, satélites, monitoramento médico, controle de processos industriais e robótica.

Na busca por uma maior redução ou até mesmo a completa ausência de pausas, foi criada uma versão do algoritmo de coleta concorrente e incremental com contagem de referências, onde a base de contagem é dada a partir da barreira de escrita (código que deve ser executado sempre que uma alteração no valor de ponteiros for realizada) e com agrupamento de objetos por regiões de memória.

Os métodos de leitura e escrita de barreira já citados por Jones e Lins (1996) são usados por coletores concorrentes para preservar a consistência dos programas em condições de corridas entre o mutador e o coletor (PIRINEN, 1998; VECHEV; BACON, 2004). Tudo que, *threads* do programa do usuário fazem é alterar o grafo de objetos na memória. Por esse motivo, é tradicional chamar essas *threads* de mutadores (DIJKSTRA et al., 1978).

O método de contagem de referências em seu modelo tradicional foi apresentado por Collins (1960), Jones e Lins (1996) e ainda é bastante usado por linguagens como Python,

Objective-C e em pesquisas (FÉREY; SHANKAR, 2016; HARVEY-LEES-GREEN et al., 2017). Além disso, a barreira de escrita também pode implicar em mais trabalho do mutador, maior custo de trabalho e tempo na compilação e execução (VECHEV; BACON, 2004). Estudos como Hellyer, Jones e Hosking (2010) e Vechev e Bacon (2004) também mostram que barreiras de escrita são necessárias para manter a consistência do programa mesmo produzindo eventuais novas coletas.

De acordo com o que foi exposto, justifica-se o uso de barreiras de escrita não somente para guardar informações sobre objetos remanescentes que não devem ser coletados, mas como um método de contagem de referência, reduzindo a necessidade de varreduras no *heap* em busca da contagem de referências a objetos alocados. Por isso, pode-se utilizar da barreira de escrita para contagem de referências.

Outro método utilizado na composição dessa nova abordagem de contagem de referências é um princípio da técnica de alocação por região de memória (DAVIS et al., 2012; DAVIS, 2015).

A proposta apresentada nesta tese visa atender à necessidade de sistemas robustos, escaláveis com acesso massivo e de tempo real. Para isso, propõe-se uma nova versão do algoritmo de contagem de referências com uso da barreira de escrita para contagem de referências e o não uso de contadores para todos os objetos no *heap*, reduzindo o espaço de memória requerido por essa técnica bem como diminuindo o número de paradas para coleta de dados não mais utilizados. Assim, esta tese objetiva analisar o gerenciamento automático e eficiente da memória, utilizando algoritmos baseados na técnica de contagem de referência e técnicas de alocações por regiões de memória, mantendo o desempenho e reduzindo o número de paradas.

## 1.1 OBJETIVOS

O objetivo deste trabalho é propor uma otimização para o problema da coleta de lixo por meio da técnica baseada em contagem de referências, incluindo um novo método de criação/limpeza de objetos em memória dinâmica em aplicações concorrentes e de tempo real. Para alcançar este objetivo são propostas melhorias nos algoritmos de Jones e Lins (1996), Lins e Jones (1993) e Formiga (2011) envolvendo todas as etapas do processo de coleta.

Assim como na linguagem Go, a abordagem aqui proposta também faz alocações por segregação de tamanho para evitar a necessidade de coletas em objetos isolados. A abordagem proposta difere das demais da literatura no algoritmo de contagem de referências, inserindo barreiras de escrita com base em análise de código durante a compilação, permitindo a redução de grandes varreduras na *heap*.

### 1.1.1 Questões de Pesquisa

Um conjunto de perguntas de pesquisa nortearam as hipóteses presentes neste trabalho:

1. Qual o impacto no tempo de execução da introdução de pausas na coleta por contagem de referência para detecção de estruturas em ciclo?
2. É possível eliminar ou reduzir o número de pausas para ação da coleta de lixo?
3. Qual o impacto no desempenho com a redução das operações nos contadores em objetos no *heap*?
4. É possível reduzir o número de varreduras na detecção de estruturas cíclicas mantendo o desempenho da contagem de referências?

Ao longo do trabalho são descritos processos e evidências experimentais que buscam responder a estas perguntas.

## 1.2 MOTIVAÇÃO

O desenvolvimento de *software* é um grande desafio, desde a escolha da linguagem a ser utilizada, planejamento e modelagem, datas de entregas e prazos, todos esses fatores causam grandes impactos no programa resultante. Esse desafio pode ser agravado em casos em que a linguagem utilizada para desenvolvimento delega ao programador a tarefa de gerenciar a memória necessária para que a computação ocorra. Nem sempre foi assim. Acreditava-se que a coleta de lixo era um consumidor desnecessário de recursos, causando atrasos no processamento. Além disso, esse mecanismo é discriminado por alguns desenvolvedores de software.

Experimentos como os apresentados em Rovner (1985) mostram que uma proporção considerável do tempo de desenvolvimento de *software* é gasto em *bugs* referentes a problemas de gerenciamento de memória, o que também é observado em plataformas móveis (MARTIE et al., 2012).

Atualmente, o gerenciamento dinâmico de memória possui uma longa e respeitável história a partir do seu surgimento para a linguagem LISP, em 1960. Ademais, o uso de coletores de lixo visa reduzir a responsabilidade do desenvolvedor e aumentar a confiabilidade do *software* em busca da reutilização de espaços de memória que não seriam mais utilizados, sem modificar o comportamento do programa ou requerer a intervenção do programador.

Durante décadas tem sido discutido o uso de muti-processadores nas diversas arquiteturas vigentes. Computadores, *smartphones*, dispositivos móveis utilizados na Internet of Things (IoT), e outros aparelhos têm tirado proveito desses avanços, porém o uso dos recursos computacionais clamam cada dia mais por maior poder de processamento e alta disponibilidade.

A melhor utilização dos recursos como concorrência, paralelismo e uso da memória estão interligados ao sistema de coleta de lixo, que têm um papel fundamental nesse cenário. Essas aspectos são discutidas a partir do gerenciamento automático de memória.

## 1.3 PRINCIPAIS CONTRIBUIÇÕES

Ao longo do texto são apresentadas otimizações e melhorias em diversos aspectos no que tange ao tópico de alocamento e liberação de memória de forma automática. As principais contribuições desta tese são:

- Melhorias propostas no sistema de alocamento de memória de forma dinâmica, baseada no uso da memória inutilizada;
- Otimizações para o sistema de coleta por contagem de referências e para detecção de estruturas em ciclo;
- A proposta da combinação das diferentes técnicas de coleta (contagem de referências e coleta por regiões de memória);
- Definição e construção de um ambiente para execução e replicação dos experimentos.

## 1.4 ORGANIZAÇÃO DA TESE

Esta tese se encontra organizada em seis capítulos, incluindo este de Introdução. O Capítulo 2 apresenta a revisão da literatura sobre o gerenciamento de memória, mostrando o histórico e as técnicas difundidas do gerenciamento automático de memória, sendo descritas as principais vantagens e desvantagens de cada uma.

No Capítulo 3, é apresentada a linguagem Golang, que foi a linguagem escolhida para validar as hipóteses propostas e também serve de parâmetro como uma das linguagens da atualidade mais utilizadas e de auto desempenho com grande adesão nos aspectos industriais e acadêmicos.

No Capítulo 4, são apresentadas as otimizações e melhorias nas técnicas de contagem de referência, alocação de memória, análise de fuga, sendo proposta uma nova abordagem do uso da técnica de contagem de referências com base em regiões de memória.

No Capítulo 5, são descritos os ambientes utilizados, bem como os testes e experimentos feitos e ainda são apresentados os resultados obtidos.

Por fim, no Capítulo 6, são apresentadas as considerações finais e trabalhos futuros.

# 2 GERENCIAMENTO DE MEMÓRIA E COLETA DE LIXO

O gerenciamento de memória não somente trata da coleta de objetos não mais utilizados, mas o local e a maneira como são alocados e influenciam na execução dos programas. Há três maneiras de alocar dados para execução de programas: alocação estática, alocação em pilha e alocação no *heap*.

A alocação estática determina as posições de memória de todos os itens que serão usados em tempo de compilação, proporcionando eficiência e segurança a execução do programa no que diz respeito à utilização de memória, permitindo acesso a dados diretamente. Essa técnica é muito utilizada em sistemas embarcados, dadas as restrições desse tipo de sistema. As restrições são primordialmente sobre segurança dos dados, evitando falhas na execução. Além disso, linguagens como C e Java (WELLINGS, 2004) utilizam essa estratégia para definir atributos globais e estáticos. Porém, a alocação estática também apresenta limitações. Todos os itens que são alocados devem definir o tamanho das estruturas em tempo de compilação, sendo que tais itens não podem crescer, limitando as possibilidades de uso do sistema.

A pilha é um espaço determinado da memória que agrupa procedimentos de um programa em execução. Cada chamada à pilha caracteriza um novo item, quando o procedimento é finalizado, os elementos são desempilhados e seu espaço é reutilizado. Nesse procedimento, o compilador precisa calcular o número e tamanho das variáveis que serão utilizadas. A pilha possui um limite de crescimento, podendo variar de programa para programa. Além disso, chamadas recursivas podem rapidamente estourar a pilha.

O *heap* é uma área de memória destinada a fazer alocações de forma dinâmica, flexibilizando o uso da memória durante a execução dos programas. Muitas linguagens de alto nível permitem o uso de alocação dinâmica em *heap*. Sendo o uso de pilhas e *heap* as abordagens mais utilizadas para o uso de alocação dinâmica em *heap*, o gerenciamento dos objetos que darão espaço para novos elementos é necessário. Esse método, como supracitado, é chamado de coleta de lixo. Linguagens como Java, C#, Python, Ruby, Go, Erlang e outras (TIOBE, 2019) usam o gerenciamento automático de memória.

O mecanismo comum a todas as técnicas de gerenciamento de memória é a identificação dos objetos que permanecem em uso para serem mantidos em memória, enquanto todos os objetos que não são mais utilizados, são desalocados dando espaço a novos objetos.

Este capítulo apresenta os principais conceitos e notações utilizadas sobre o gerenciamento de memória, onde são descritas as técnicas de coleta de lixo, com foco nas vantagens e desvantagens das três técnicas de gerenciamento dinâmico de memória.

## 2.1 GERENCIAMENTO DINÂMICO DE MEMÓRIA

Um dado alocado dinamicamente no *heap* é chamado **objeto**, **célula** ou **nó**. Objetos no *heap*, estando vinculados aos registradores do processador, aos da pilha do programa ou às variáveis globais, contêm referências aos dados da *heap* na forma de **raízes** (***roots***).

Uma célula alocada individualmente no *heap* denomina-se alcançável, se seu endereço está vinculado a um nó raiz ou um ponteiro que vincule esse nó a outro que também está vivo dentro do *heap*. Objetos no *heap* que não são mais usados, ou seja, que não podem mais ser acessados por nenhuma variável em execução no programa, são denominados **lixo**, e podem ser desalocados. Nem sempre é possível detectar se um objeto está ativo, ou seja, se há um caminho que conecta diretamente esse termo ao conjunto raiz do programa (ANDREW, 1998).

Em alguns algoritmos e técnicas de coleta de lixo é utilizada a abordagem conservativa, que busca fazer a análise de vida dos objetos alocados com ajuda de algoritmos específicos, utilizando principalmente cálculos e informações dos ponteiros no fluxo de controle em tempo de execução.

Como descrito na seção anterior, o gerenciamento de memória dinâmica busca sanar o problema de quando liberar espaços de memória com objetos que não são mais utilizados. O conjunto de células que estão disponíveis para o uso é denominado **células livres** (***free-list***).

Seguindo a terminologia **mutador** é a aplicação do usuário, cujo papel é alterar ou mutar o grafo de conexões durante o ciclo de coleta de dados ativos no *heap*. É caracterizado um **ciclo** quando um grupo de objetos faz referência a um mesmo grupo, onde, algum deles pode não mais ser acessível pela raiz do programa. Esses conceitos são utilizados no gerenciamento manual e dinâmico. Além disso, com o surgimento das primeiras linguagens, como Fortran (BACKUS, 1958), Cobol (MACKINSON, 1961) e Algo 58 (PERLIS; SAMELSON, 1958) tal tarefa era de responsabilidade do desenvolvedor da aplicação.

Ao longo da história da computação, surgiram técnicas para a automatização dessa tarefa, sendo a coleta por marcação e varredura, contagem de referências e coleta por cópia as três principais técnicas utilizadas até hoje e são base para todas as demais técnicas do segmento (JONES; LINS, 1996).

### 2.1.1 Algoritmos sequenciais

Os algoritmos de coleta de lixo nesta seção foram criados para ambientes uniprocessados, ou seja, com um único processo, onde o mutador para o processo do usuário a fim de que o coletor entre em ação (WILSON, 1992).

#### 2.1.1.1 Marcação-varredura (*Mark-Sweep*)

O primeiro algoritmo de reciclagem de memória dinâmica foi baseado na técnica de rastreamento, também chamado de *Mark-Scan*, criado por J. McCarthy em 1960 para a linguagem Lisp (MCCARTHY, 1960).

Essa abordagem baseia-se no rastreamentos de objetos a partir do conjunto raiz, seguindo os ponteiros de cada objeto ligado, caracterizando o processo de marcação de objetos. Posteriormente, é feita a varredura de todo o *heap* que não estão mais marcados.

Mediante uma nova alocação em que, se é identificado que não há mais espaço livre, é iniciado o processo de liberação de memória. Nesse momento, o sistema pode ficar em suspensão, para que haja varredura de todos os objetos inativos (JONES; LINS, 1996).

O Algoritmo 1 exemplifica a alocação de uma nova célula no *heap* e a vincula ao grafo, obtendo uma nova célula do conjunto de células livres. O momento de realizar a varredura é iniciado quando o programa requer a alocação de uma nova célula e não há mais memória disponível. Esse ciclo de coleta suspende o processo do usuário até que a coleta seja finalizada.

**Algoritmo 1:** Alocação de uma célula no Algoritmo de *Mark-Sweep*.

```
1 Function New() =
2     if free_pool is empty then
3         mark_sweep()
4     newcell = allocate()
5     return newcell
```

A coleta é dada por duas fases, como mostra o Algoritmo 2. A primeira fase é a de marcação, em que as células vivas são percorridas e marcadas. Na segunda fase, o *heap* é percorrido para identificação das células não marcadas.

**Algoritmo 2:** Algoritmo de coleta de lixo no *Mark-Sweep*.

```
1 Function mark_sweep() =
2     for R in Roots do
3         mark(R)
4     sweep()
5     if free_pool is empty then
6         abort "Memory Exhausted"
```

A marcação das células usa um *bit* por célula (*mark-bit*). Esse *bit* é utilizado para registrar a acessibilidade das células a partir da raiz. Assim, todo o grafo é marcado por profundidade (*depth-first*) (ver o Algoritmo 3).

A fase de varredura está detalhada no Algoritmo 4. A varredura é responsável por recolher o lixo, onde todos os objetos vivos já são distinguíveis dos objetos que são lixo.

**Algoritmo 3:** Marcação por recursão

```
1 Function mark(N) =
2     if mark_bit(N) == unmarked then
3         mark_bit(N) = marked
4         for M in Children(N) do
5             mark(*M)
```

Assim, todo o *heap* é varrido linearmente e os objetos livres podem ser reaproveitados, ou seja, incluídos na lista livre novamente.

**Algoritmo 4:** Algoritmo de varredura

```
1 Function sweep() =
2     N = Heap_bottom
3     while N < Heap_top do
4         if mark_bit(N) == unmarked then
5             free(N)
6         else
7             mark_bit(N) = unmarked
8         N = N + size(N)
```

Esse algoritmo tem o custo proporcional ao número de nós marcados, e ao tamanho do *heap*. Nessa abordagem, quando se deseja alocar um item de tamanho n, porém onde todas as posições livres de memória são de tamanho inferior a n (fragmentação externa), força a coleta de dados, onde o sistema pode passar um tempo arbitrário em pausa.

Também é possível que ocorra fragmentação interna, que é caracterizada quando o objeto é alocado em uma posição com tamanho superior ao valor do objeto, ocupando regiões de memória em que restam espaços livres mal utilizados. Ainda assim, essa abordagem pode oferecer um alto desempenho quando comparada às outras técnicas de coleta.

Uma problemática que pode repercutir no desempenho da coleta por marcação e varredura é que essa técnica depende do espaço ocupado no *heap*: quanto mais ocupado o *heap*, maior o custo computacional de recolher células. Isso pode impactar diretamente no tempo de execução do programa dado o tempo de suspensão durante as coletas.

#### 2.1.1.2 Contagem de Referências

A técnica de contagem de referência se destacou entre as demais por ser considerada simples, direta e incremental. Esta técnica realiza a guarda do objeto alocado juntamente com o número total de referências, possibilitando varreduras ou buscas em grafos, vantagem primordialmente utilizada para os sistemas de tempo real.

Cada referência é atualizada à medida que são adicionadas novas referências ao objeto. Quando o valor de referência de um objeto chega a zero, essa mesma célula pode dar espaço a um novo objeto (JONES; LINS, 1996). A performance da contagem de referências é proporcional ao tamanho de memória ocupada, onde, a carga de gerenciamento é distribuída de forma não uniforme.

O Algoritmo 5 mostra a alocação de novas células na Contagem de Referência (CR). As células são removidas da *free-list* e ligada ao conjunto raiz do programa e cada célula pode manter seu próprio contador. O contador de referências permanece inalterado para uma célula obtida da *free-list* (RC = 1).

**Algoritmo 5:** Alocação de células do Algoritmo de contagem de referências.

```
1  Function allocate() =
2      newcell = free_list
3      free_list = next(free_list)
4      return newcell
5  Function new() =
6      if free_list == nil then
7          abort "Memory Exhausted"
8      newcell = allocate()
9      RC(newcell) = 1
10     return newcell
```

Os métodos *free*, *update* e *delete*, apresentados no Algoritmo 6, também operam os ponteiros das células, ou seja, suas referências, isso para garantir o invariante da contagem. *Free* é chamado pelo método *delete* para mover uma célula inativa para a *free-list*; *delete* é usada quando a referência de uma célula é destruída, ou será decrementada para zero; enquanto *update* atualiza a referência da célula, toda vez que um novo ponteiro aponta para a célula.

Essa estratégia permite que objetos sejam liberados de maneira imediata, reaproveitando a memória assim que o objeto não estiver mais em uso. Esse procedimento minimiza a paginação na memória.

Embora a contagem de referências seja atrativa por ser simples, ela apresenta problemas, tais como:

1. Objetos que se apontam entre si, formando ciclos, não podem ser reciclados mesmo que os objetos já não sejam mais acessíveis, pois seus contadores continuam com valor igual a 1.

2. A operação de incremento e decremento na contagem de referência tem um alto custo por tratar de muitos objetos na *heap*, aumentando significativamente o total de operações realizadas.

**Algoritmo 6:** Atualização dos contadores na contagem de referências.

```
1  Function free(N) =
2      next = free_list
3      free_list = N
4  Function delete(T) =
5      RC(T) = RC(T) - 1
6      if RC(T) == 0 then
7          for U in Children(T) do
8              delete(*U)
9          free(T)
10 Function Update(R, S) =
11     RC(S) = RC(S) + 1
12     delete(*R)
13     *R = S
```

Significantes melhorias foram propostas para o algoritmo de contagem de referências, tais como evitar a contagem para objetos e variáveis em pilha (BACON et al., 2001; WASSERMAN, 2013). Também foram propostas outras abordagens especiais para a problemática de estruturas cíclicas como varreduras locais (LINS, 1992a; LINS, 2008).

O problema de contagem de referências cíclicas causa baixa adesão da técnica em meio às linguagens. Ciclos de objetos inativos que se referenciam impossibilitam que a contagem chegue a zero, portanto, os objetos nunca são liberados. Nesta tese, será implementado um dos algoritmos de CR com detecção de estruturas cíclicas para sistemas multiprocessados apresentado por Formiga (2011), com a finalidade de melhorar o uso da memória e manter a precisão de coletas em ciclo.

#### 2.1.1.3 Coleta por Cópia

A primeira versão desse algoritmo foi proposta por Marvin Minsky em 1963 (MINSKY, 1963), que copiava células vivas do *heap* para uma área de armazenamento secundário, logo após revertia de volta para a mesma *heap*, ou seja, usando a segunda área como armazenamento temporário, na qual era necessário o uso de estruturas auxiliares para guardar o endereço da nova posição de memória.

Esse algoritmo voltou aos holofotes com o advento dos sistemas operacionais com memória virtual, Fenichel e Yochelson (1969) propuseram o uso do algoritmo de cópia, para armazenamento em memória interna.

A inicialização para a coleta por cópia é ilustrada no Algoritmo 7. Tradicionalmente, os semiespaços recebem os nomes de espaço origem *from space* e espaço destino (*to space*), onde *free* aponta para o primeiro endereço livre para alocação.

A coleta por cópia também é considera uma técnica de rastreamento. Pois o *heap* é

**Algoritmo 7:** Inicialização e alocação na coleta por cópia.

```
1  Function init() =
2      To_space = Heap_bottom
3      space_size = Heap_size / 2
4      top_of_space = To_space + space_size
5      From_space = top_of_space + 1
6      free = To_space
7  Function New(n) =
8      if free + n > top_of_space then
9          flip()
10     if free + n > top_of_space then
11         abort ("Memory Exhausted")
12     newcell = free
13     free = free + n
14     return newcell
```

dividido em 2 grandes regiões e todos os objetos são alocados no primeiro segmento da memória. Quando não há mais espaço para uma nova alocação, é feita uma varredura que atravessa todo o grafo e as células em uso são copiadas para o segundo segmento da memória pela rotina *flip()* (ver Algoritmo 8), por fim, o restante é mantido para coleta (JONES; LINS, 1996).

**Algoritmo 8:** Rotina da cópia entre os espaços.

```
1  Function flip() =
2      From_spave, To_space = To_space, From_space
3      top_of_space = to_space + space_size
4      free = To_space
5      for R in Roots do
6          R = copy(R)
```

Na abordagem de coleta por cópia, os dois semi-espaços chamados de *from space* e *to space* trocam de papel a cada ciclo de coleta. Cada objeto copiado é deixado em um endereço de encaminhamento que aponta para o novo endereço. Assim, durante o procedimento de cópia, todos os objetos visitados são verificados se já não foram copiados pelo algoritmo. A versão deste pseudocódigo apresentado no Algoritmo 9 foi desenvolvido por Fenichel e Yochelson (1969).

O valor de *P'* aponta para uma palavra, não uma célula. Cada endereço de encaminhamento é mantido na própria célula, não sendo necessário uso de espaço adicional. Devido à troca de regiões de memória a cada ciclo de coleta, o problema de fragmentação é eliminado, mas a redução da região total do *heap* pode causar um alto custo em sistemas que fazem uso de grandes quantidades de memória (JONES; LINS, 1996).

**Algoritmo 9:** Algoritmo de coleta por cópia.

```
1  Function copy(P) =
2      if atomic(P) or P == nil then
3          return P
4      if not forwarded(P) then
5          n = size(P)
6          P' = free
7          free = free + n
8          temp = P[0]
9          forwarding_address(P) = P'
10         P'[0] = copy(temp)
11         for i = 1 to n - 1 do
12             P'[i] = copy(P[i])
13     return fowarding_address(P)
```

A Tabela 1 a seguir mostra as vantagens e desvantagens das três técnicas:

| | **Vantagens** | **Desvantagens** |
|---|---|---|
| *Mark-Sweep* | • Utiliza toda a *heap*<br>• Não requer mais que 1 únio bit para marcação e varredura<br>• Não faz operações de ponteiros | • Requer parada total da execução do programa para coleta<br>• Precisa varrer diversas vezes toda a *heap*<br>• Apresenta erros de fragmentação |
| Contagem de Referência | • Interopera com a execução do programa, requerendo pequenos passos para o funcionamento<br>• Não depende do tamanho total da memória | • Apresenta problemas com estruturas cíclicas<br>• Alto custo para manter referências atualizadas<br>• Requer mais espaço para o contador de cada item |
| Cópia | • Evita fragmentação movendo itens entre as regiões de memória | • Alto custo de cópia entre os espaços de memória<br>• Grande número de pausas para mover objetos entre as regiões de memória<br>• Pode apresentar problemas de perda de referência dos objetos, devido à movimentação entre regiões de memória (*motion sickness*) |

Tabela 1 – As três principais técnicas de gerenciamento automático de memória.

#### 2.1.1.4 Coleta de lixo geracional

A coleta por cópia despende um grande espaço copiando células em uso entre os semi-espaços a cada coleta. Também foram observados em estudos empíricos que, em muitos programas, os objetos recém-criados tendem a ser coletados nos primeiros ciclos de coleta (FORMIGA, 2011), enquanto objetos que sobrevivem após alguns ciclos tendem a permanecer ao fim da execução. Com isso, coletores concentram esforços em elementos recém-criados (JONES; LINS, 1996).

A partir disso, a técnica de coleta conhecida como coleta Geracional foi concebida, na qual objetos são segregados por gerações, permitindo que cada geração de objetos alocados seja coletada em frequências diferentes. Por exemplo, objetos antigos em menor frequência que objetos recém-alocados, podendo até alcançar gerações que nunca sofrem coleta. Muitas vezes essa abordagem é vista como uma junção das três técnicas supracitadas (MARLOW et al., 2008). ANDREW (1998) afirma que cerca de 10% dos objetos recém-alocados não passam de um ciclo de coleta.

Diferentes abordagens podem ser desenvolvidas com esse tipo de coleta, pois $n$ diferentes gerações de objetos podem ser mantidas (onde $n \geq 2$).

Também há um custo alto em manter a informação dos dados das gerações antigas, bem como se o programa realiza mais atualizações do que novas alocações, isso pode gerar um elevado custo para a gerações de objetos. Assim temos:

- Novos objetos são sempre alocados na geração mais jovem;
- Gerações jovens são coletadas com maior frequência;
- Quando se dá uma coleta, outras gerações mais jovens podem ser afetadas;
- Objetos podem ser promovidos entre gerações quando sobrevivem as coletas.

### 2.1.2 Coleta Incremental

A coleta de lixo pode representar uma baixa percentagem do tempo total de computação de um programa, porém, essa interrupção pode variar de forma significativa, e isso é indesejado, principalmente para sistemas de tempo real. Desse modo, coletas concorrentes e incrementais podem trazer grandes benefícios aos programas que utilizam gerenciamento de coleta dinâmica.

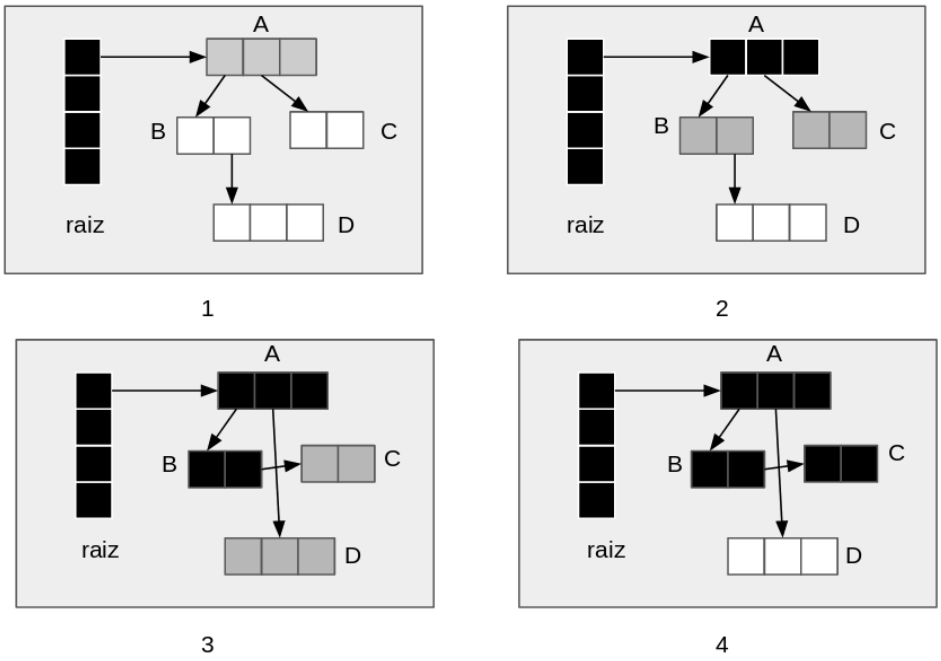
Técnicas de coleta têm dois fundamentos principais: a identificação de objetos que não podem mais ser alcançáveis pelo programa e reutilização desses espaços de memória. A coleta incremental realiza pequenas operações concorrentemente à execução do programa para evitar o grande número de pausas, diferentemente das técnicas tradicionais de rastreamento que realizam todo o ciclo de coleta em um único passo.

#### 2.1.2.1 Marcação por três cores (*Tri-color marking*)

Um exemplo desse tipo de coleta é o chamado *Tricolor marking*, que define três passos por cores diferentes para a coleta de forma incremental, sendo **branco**, **cinza** e **preto** (DIJKSTRA et al., 1978).

Um objeto em branco indica que ele ainda não foi visitado; objetos em cinza determinam que esses objetos já foram visitados (marcados ou copiados), mas seus nós filhos ainda não foram visitados; e objetos da cor preta representam objetos que já foram marcados bem como seus filhos. Assim, iniciando todos com a cor branca e passando pela cinza e por última a preta, quando todos são visitados, não havendo mais objetos da cor cinza, todos os objetos podem ser coletados.

Figura 1 – Exemplo do processo de marcação do algoritmo *Tri-color* de Dijkstra.




**Fonte:** o autor.

Ainda na primeira fase, o algoritmo *Tri-color* marca todos os objetos como branco, logo os objetos são escaneados e todos objetos alcançáveis, variáveis globais e locais, são marcados de cinza, objetos que eram alcançáveis antes do processo de limpeza que perdem sua referência ao término dele são retidos para o próximo ciclo de coleta, sendo tanto a fase de marcação e varredura concorrentes com a aplicação, a fase de marcação encontra todos os objetos que são alcançáveis para marcá-los (como ilustrado na Figura 1), deixando o restante dos objetos sem marcação. Durante a varredura, passa-se por todo o *heap* onde são encontrados todos os objetos remanescentes sem marcação; todos esses são adicionados na *free-list*, na segunda fase objetos que estão marcados como cinza serão marcados de preto, nenhum objeto marcado como preto pode apresentar referência

alguma aos objetos marcados como branco. O mesmo ocorre entre as marcações de cinza para preto. O processo se repete até que não haja mais objetos em cinza, assim todos os objetos já não são mais alcançáveis e o espaço de memória pode ser reutilizado.

Esse tipo de técnica pode se adequar a qualquer tipo de coletor, sendo encontrada em diversas linguagens. As abordagens de algoritmos incrementais e concorrentes tiram proveito do uso do mutador para finalizar as tarefas realizadas.

#### 2.1.2.2 Contagem de referências paralelo

Há algoritmos de Lins desenvolvidos para máquinas paralelas fortemente acopladas. Lins e Jones (1993) propuseram um algoritmo de CR ponderada capaz de identificar estruturas cíclicas em ambientes de processamento distribuídos. Foi identificado que este algoritmo não é totalmente concorrente, podendo bloquear a coleta enquanto espera que um outro processo finalize sua operação. Esta deficiência foi corrigida por Jones e Lins (1993) no mesmo ano. A solução proposta por Lins adiciona barreiras de sincronização a cada fase da marcação e varredura, garantindo que as coletas de lixo simultâneas não interfiram entre si, não perdendo a cor e a informação do peso.

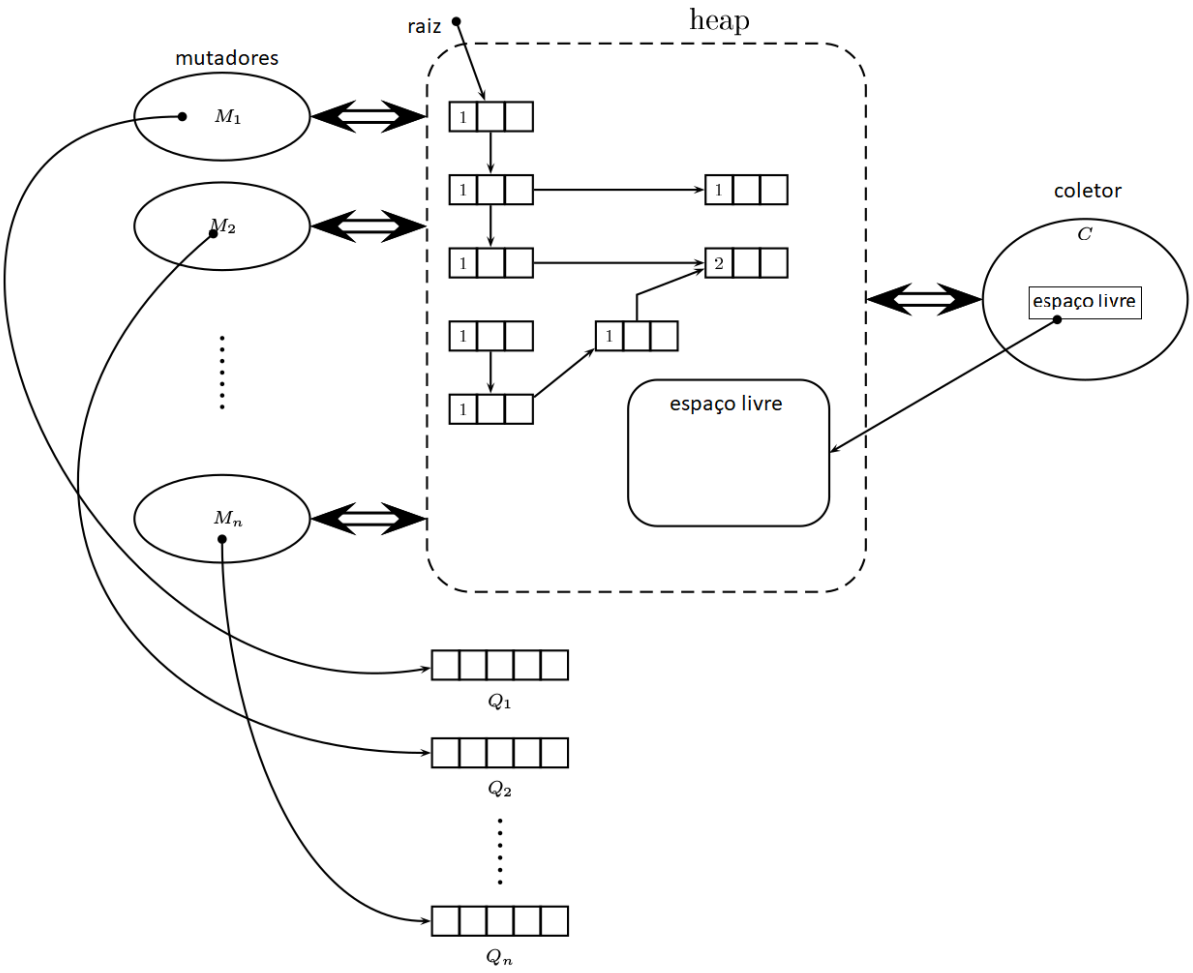
Formiga e Lins (2007) estenderam o algoritmo para o uso de vários mutadores, fazendo com que os mutadores compartilhem acesso a registradores que guardam referências para o topo da lista livre, como ilustra a Figura 2.

A Figura 2 mostra como o algoritmo trabalha utilizando vários mutadores, os quais são representados por $M_1$, $M_2$, ..., $M_n$. Pode-se identificar uma *thread* pelo seu identificador único. É preciso sincronizar todas as filas de atualização, sem parar todos os mutadores ao mesmo tempo.

Além disso, são usadas *flags* para coordenar a ação do mutador e coletor. A *flag collect* foi usada para indicar que um novo ciclo de coleta foi iniciado. Um mutador insere um marcador na sua fila de atualização assim que verifica a presença da *flag collect* isso indica o ponto onde as filas são sincronizadas. Cada mutador $i$ também insere um marcador assim que o marcador já tiver sido inserido na fila $Q_i$. Logo após, o coletor poderá obter os registros de atualização de cada pilha até encontrar o marcador.

A versão mais atualizada desse algoritmo foi demonstrada em Formiga (2011).

Figura 2 – Exemplo da organização do algoritmo para um coletor e vários mutadores.




**Fonte:** Formiga e Lins (2007).

# 3 GERENCIAMENTO DE MEMÓRIA EM GO

Nesse capítulo, são apresentadas as características da linguagem Go. Essa linguagem é utilizada para a base de estudos das melhorias e abordagens apresentadas nesta tese.

Go teve início como projeto interno da Google e foi inspirado por muitas outras linguagens como: C, Pascal, Aleg e Oberon. A sintaxe é não-ambígua e não-intrusiva, enquanto ainda mantém informações suficientes para o compilador executar uma verificação de tipo forte do código-fonte. Go é uma linguagem procedural, imperativa, estaticamente tipada, de propósito geral (*Web*, *desktop*, sistemas embarcados e outros), provê suporte à programação concorrente por meio de canais (*channels*), *interfaces*, criação de tipos e funções de alta ordem (PIKE, 2009; DONOVAN; KERNIGHAN, 2015). Go possui uma comunidade de desenvolvedores ativa com bibliotecas disponíveis de variados serviços. Além disso, Go foi escolhida principalmente por seus métodos de tratamento a milhões de acessos simultâneos ou em tempo real.

Go está na décima quinta posição no TIOBE (TIOBE, 2019) entre as linguagens mais utilizadas de 2018, décima quarta na lista de linguagens do RedMonk (REDMONK, 2018), que indexa as linguagens com base nos repositórios do GitHub e perguntas do StackOverFlow. Além disso, está em primeiro lugar na lista em linguagens para aprender de Pe-Than, Dabbish e Herbsleb (2018). Outros segmentos podem ser encontrados em (VALKOV; CHECHINA; TRINDER, 2018).

Implementando o modelo de concorrência de Hoare (HOARE, 1978), Go permite suporte à coleta de lixo dinamicamente, permitindo que o desenvolvedor não precise alocar ou desalocar memória manualmente. Embora o coletor de Go possa retardar a coleta para alguns dados definidos pelo programador, ele utiliza o algoritmo de marcação e varredura *Tri-color* (JONES; LINS, 1996). Este coletor faz uso de pausas nos processos em execução para coleta de lixo, muitas vezes essas pausas são indesejadas, principalmente quando se trata de sistemas de tempo real. É sabido que essas pausas consomem tempo por percorrer a *heap* inteira e ainda podem apresentar erros de fragmentação.

A linguagem Go reduziu as pausas rodando o GC concorrentemente entre os processos em execução. Hudson (2015) reporta que a coleta em Go registrou pausas abaixo de 10 milissegundos, o que já pode ser suficiente para suportar programas de tempo real, como jogos e comércio (HUDSON, 2015).

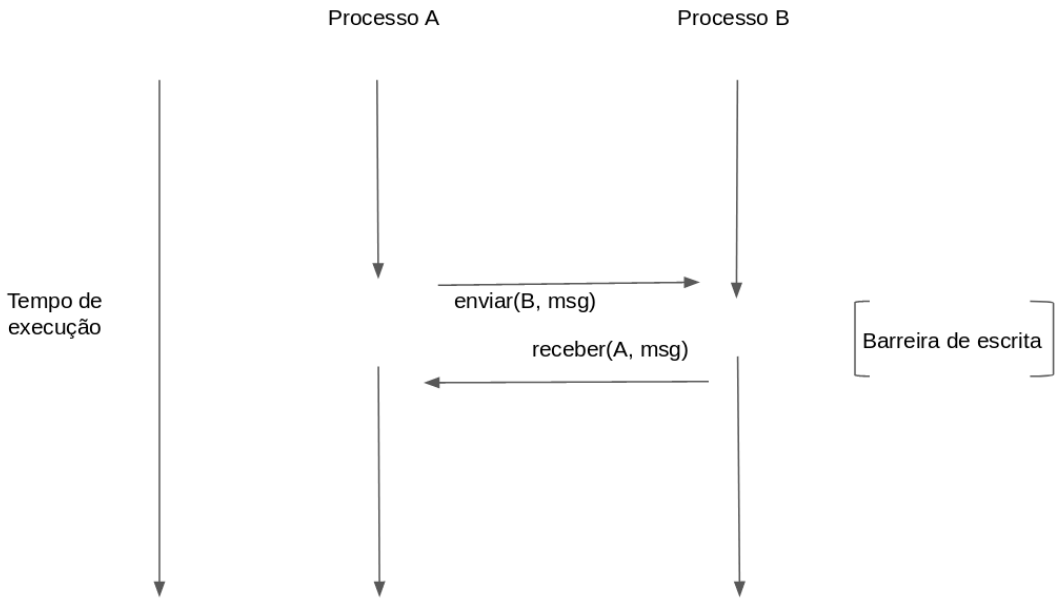
## 3.1 O MODELO DE CONCORRÊNCIA EM GO

Linguagens de programação tradicionais, principalmente orientadas a objetos (assim como Java), utilizam o modelo de memória baseada em memória compartilhada. Isso significa que os processos compartilham uma única memória *heap* para manipular os objetos. Essa

característica é necessária para que todos os processos tenham acesso aos dados do *heap*. As linguagens orientadas a objetos apresentam o modelo de concorrência baseado em memória compartilhada, onde todos os *threads* podem acessar simultaneamente a memória, porém isso pode gerar inconsistência nos dados presentes na memória, sendo o desenvolvedor responsável por usar mecanismos para manter a conformidade na ação de cada processo.

As primitivas de concorrência de Go (*goroutines* e *channels*), onde os *channels* são usados para passar referências a dados entre *goroutines*, ou seja, os processos compartilham dados comunicando-se uns com os outros. Esse modelo é conhecido como CSP (HOARE, 1978). O CSP define a composição de unidades comportamentais independentes que se comunicam entre si, sendo cada unidade independente, podendo ser formada por unidades menores, que, por sua vez, são combinadas por algum padrão de interação específico (troca de mensagens). CSP permite a descrição de sistemas em termos de processos. Esses processos operam de maneira independente e interagem uns com os outros através de comunicação por troca de mensagens, como mostra a Figura 3. Nessa figura, foi descrito um exemplo de comunicação entre processos que utilizam o modelo de concorrência de CSP, onde o processo realiza uma sincronização através da barreira de escrita. Isso retira a necessidade de uso explicito de travas (*looks*) para mediação de acesso aos dados.

Figura 3 – Exemplo da execução de processos por CSP.




**Fonte:** o autor.

Em sequência, a linguagem Go garante que apenas uma *goroutine* tenha acesso aos dados em um determinado momento. Uma *goroutine* existe somente no espaço virtual do tempo de execução de Go, ou seja, não está presente no Sistema Operacional (SO).

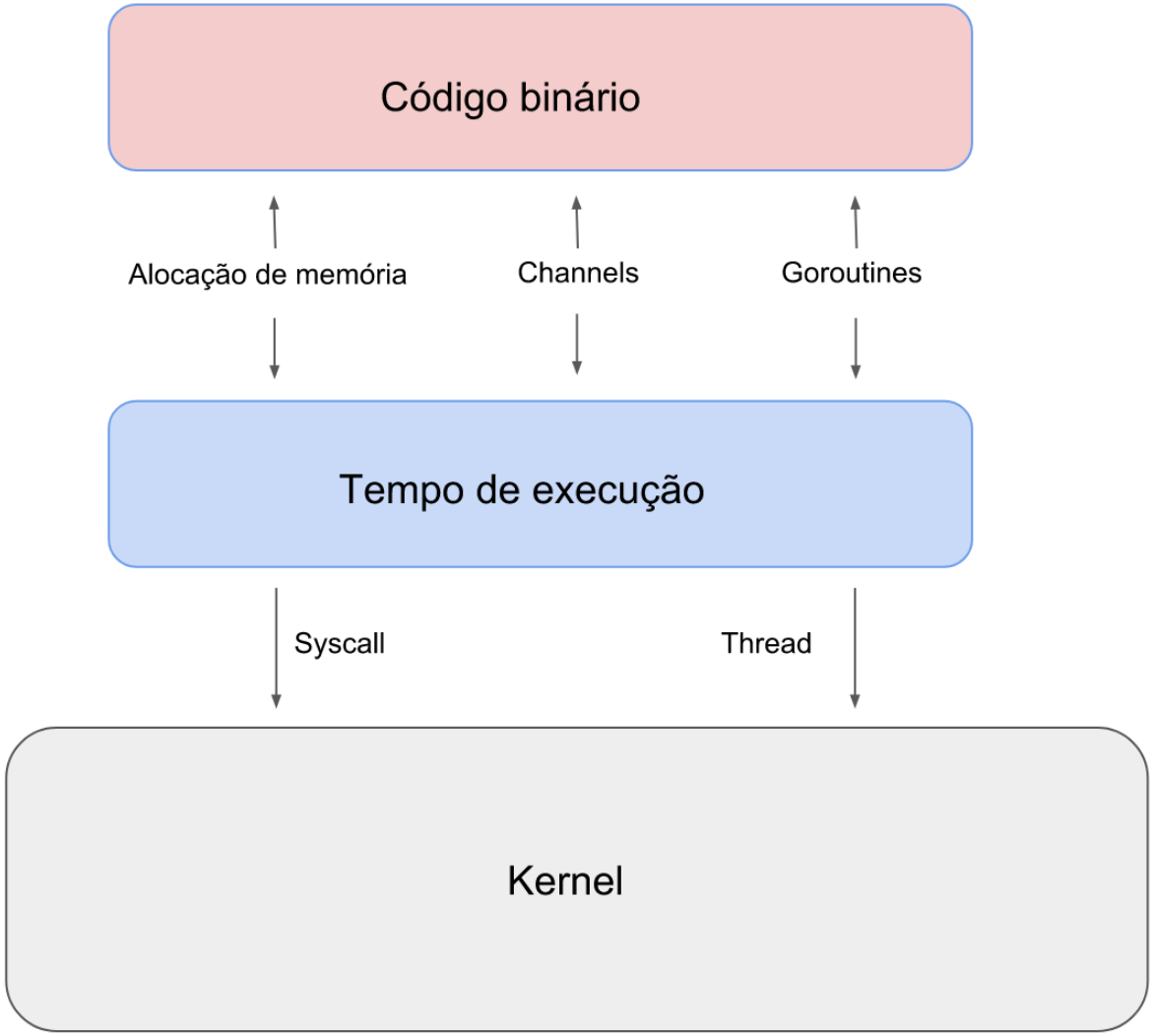
Portanto, é necessário um escalonador de tarefas de Go para gerenciar o ciclo de vida dos processos. Além disso, durante o tempo de execução de Go, são mantidas três estruturas:

- **G** que representa uma única rotina de Go, com o ponteiro da pilha, seu identificador, *cache* e *status* (ativo ou não);
- A estrutura **M**, que representa uma *thread* do SO, contendo um ponteiro para a fila de *goroutines*, bem como a *goroutine* em execução e a referência do escalonador de tarefas;
- Por última, a estrutura **Sched**, que é a estrutura global com as filas de *goroutines* e *threads*.

  Assim, na inicialização, o tempo de execução inicia uma série de *goroutines* para coleta de lixo concorrente, o escalonador de tarefas e o código do usuário.

  A Figura 4 mostra a arquitetura do tempo de execução de Go. O comando *Syscall* requisita uma nova *thread* ao SO, Go pode ter mais de uma *thread* em tempo de execução. Cada *thread* pode ter várias *goroutines*. Uma *goroutine* requer 2 Kilobyte ($KB$) de memória, enquanto *threads* podem ocupar até 1 Megabyte ($MB$) de memória, sendo que uma única *thread* pode conter milhares de *goroutines* (DESHPANDE; SPONSLER; WEISS, 2012).

Figura 4 – Exemplo do tempo de execução de Go.




**Fonte:** (DESHPANDE; SPONSLER; WEISS, 2012).

## 3.2 O COMPILADOR DE GO

Go tem ganho atenção de desenvolvedores ao redor do mundo, atualmente chegando a lançar duas novas versões da linguagem por ano, com suporte a diversas plataformas como: Linux, Windows, MacOS, OpenBSD, DragonFly BSD, FreeBSD, Solaris e outros tipos de arquitetura, e com vasto suporte a *bug-fix* entre os lançamentos.

O compilador de Go gera código binário para multiplataforma (*cross-compile*), isso significa que Go dá suporte a diversas plataformas e arquiteturas como dito anteriormente. Os desenvolvedores de Go mantêm dois diferentes compiladores para a linguagem, Gogc assim conhecida, é o compilador original da linguagem, esse define a base de todas as

ferramentas e bibliotecas escritas em Go; Go frontend (Gccgo) é a segunda opção de compilação com um foco diferenciado.

O compilador Gogc suporta os processadores, x86(32-bit e 64-bit) e ARM. Gccgo dá suporte a todos os processadores que o GNU Compiler Collection (GCC) suporta (PIKE, 2009), incluindo SPARC, MIPS, PowerPC e outros, sendo assim Gccgo um *frond-end* para o GCC. Ambos compiladores usam o mesmo tempo de execução provido por Go. Segundo os próprios criadores, Gccgo é mais lento que o original, porém dá maior suporte às otimizações de compilação, isso gera benefícios ao código gerado, de modo a apresentar uma performance diferenciada à execução dos programas, mas ambas as versões são semelhantes no que diz respeito às técnicas utilizadas, forma de alocação, e estratégias usadas para coleta de dados, essas técnicas são mostras a seguir.

## 3.3 ANÁLISE DE FUGA EM GO

Um dos fatores que torna possível a alta taxa de desempenho da linguagem Go, principalmente no que tange ao requisito de gerenciamento de memória, é o fato de Go utilizar diversas técnicas, a análise de fuga é uma delas, com base em abordagem por geração, onde objetos recentemente alocados tendem a ser coletados primeiro. Go utiliza a pilha em primeira instância para todos objetos recém alocados, ou seja, todo objeto a ser alocado parte do princípio que esse estará na pilha, logo, durante o tempo de compilação, é feita a análise de fuga dos itens presentes na pilha. Quando essa análise identifica que um objeto da pilha escapa para outros escopos (como diferentes funções e pacotes do programa), sua posição é modificada da pilha para o *heap*, tornando mais eficiente e direcionando o trabalho do coletor de lixo (HUDSON, 2015).

No lugar de passar ponteiros para uma função, ela passa o valor real, assim para todo ponteiro que passa de escopo para outro, esse valor não permanece mais na pilha e sim é adicionado no *heap*, essa análise é conhecida como análise de fuga (KOTZMANN; MÖSSENBÖCK, 2005; REIJERS; ELLUL; SHIH, 2018).

Assim, temos $x$ como sendo um nova variável que receberá o valor $T$, logo `x := new(T)`. O compilador tentará alocar, primeiramente, $x$ na pilha, porém, frequentemente isso não é possível. Por essa razão, diz-se que o valor $x$ escapou para o *heap*, portanto, dizemos que um objeto escapa para o *heap*, quando:

- $x$ é alcançado por algum outro objeto do *heap*, exemplo quando $x$ é atribuído a uma variável global;
- $x$ pode ser acessado além da função de escopo local onde é definida, a exemplo quando $x$ é usada como retorno da função;
- $x$ tem tamanho variável, exemplo quando é definido como um *array* ou estrutura de lista;

- quando o compilador não tem muita informação sobre como o valor ou seu ponteiro será usado, quando são passados como parâmetros de funções.

```
1  package main

3  func inlined() []int {
       var a [5]int
5      return a[:]
   }
7
   func no_inline() []int {
9      var b [5]int
       return b[:]
11 }

13 func main() {
       var local_array [5]int
15     var local_var int
       println(no_inline())
17     println(inlined())
       println(local_array
           [:])
19     println(&local_var)
   }
```

Código 3.1 – Código exemplo análise de fuga.

Toda a análise é feita durante a compilação. O Código 3.1 mostra um pequeno exemplo de um programa em Go com duas variáveis locais e dois métodos inlined() e no_inline(). Ambas as funções retornam à posição do *array* definido, a função principal *main* cria um *array* de inteiros e define um valor como inteiro; logo após são chamadas as funções onde são impressos os ponteiros de cada valor. Podemos verificar toda a análise feita pelo compilador usando a *flag* **-gcflags='-m'**. A saída do processo de análise é apresentada a seguir:

```
   ./1.go:3:6: can inline inlined
2  ./1.go:7:14: can inline no_inline
   ./1.go:11:22: can inline main
4  ./1.go:14:50: inlining call to no_inline
   ./1.go:15:52: inlining call to inlined
6  ./1.go:5:21: a escapes to heap
   ./1.go:4:13: moved to heap: a
8  ./1.go:9:29: b escapes to heap
   ./1.go:8:21: moved to heap: b
10 ./1.go:14:50: main b does not escape
   ./1.go:15:52: main a does not escape
12 ./1.go:16:60: main local_array does not escape
   ./1.go:17:53: main &local_var does not escape
```

O compilador decide alocar o valor de inlined.*a* da linha 5 e o valor inline.*b* no *heap* porque os valores são retornados de cada função saída do escopo interno de cada função, porém, os valores na função principal *main* a e b logo são alocados na pilha da função *main*, assim como os valores da variável local local_*array* e o ponteiro do mesmo. Go não

dá suporte a nenhuma palavra reservada para que desenvolvedores possam definir o que irá permanecer na pilha, ou deve ser alocado na *heap*. Todo o processo é feito de maneira automática.

## 3.4 BARREIRA DE ESCRITA EM GO

O compilador de Go também desempenha um papel importante em tornar a alocação e coleta de Go eficiente. Este emite escritas de barreiras durante a compilação dos ponteiros que apontam para objetos no *heap*, esse processo conhecido como barreira de escrita é descrito por Dijkstra et al. (1978).

Barreiras de escrita são mecanismos de sincronização que protegem o coletor de liberar objetos ainda em uso (VECHEV; BACON, 2004). Um exemplo é apresentado a seguir:

```
1  package foo

3  type t struct {
       x func() *t
5  }

7  func f() *t {
       g := new(t)
9      g.x = f
       return g
11 }
```

Código 3.2 – Código exemplo para barreira de escrita.

O Código 3.2 ilustra um caso em que o compilador emite a instrução de barreira de escrita, primeiramente porque o mesmo foge do escopo definido, por essa razão é alocado no *heap*, o asteristico $(*)indicaoponteiroemt$.

```
1  ./wb.go:7:6: can inline f
   ./wb.go:8:13: new(t) escapes to heap
```

O trecho a seguir mostra a linha em que é inserida a barreira de escrita em que o valor $f$ recebe a atribuição. Esse saída pode ser obtida usando a *flag* **-gcflags '-d wb'**.

```
./wb.go:9:9: write barrier
```

Barreiras de escrita podem ser implementadas tanto em hardware como em *software*, sendo implementadas em *hardware*, não necessitam de maiores instruções, porém, pode requerer hardware especial; enquanto barreiras em software são criadas para cada leitura ou escrita de todo ponteiro (JONES; LINS, 1996). No entanto barreiras de escrita implementadas diretamente em hardware são menos invasivas, quando comparadas a barreiras de escrita em software, o custo dessas implementações fica a cargo do mutador que pode ter uma sobrecarga de trabalho; outro fator é o espaço em memória necessário para gravar tais informações. Por essa razão, em Go, barreiras de escrita não ficam ativas durante toda a execução do programa, os dados dos ponteiros são gravados somente quando o coletor

está operando. Isso aumenta significativamente o desempenho da coleta (HUDSON, 2015), pois barreiras de escrita também ajudam a manter consistência de programas.

## 3.5 O COLETOR DE GO

A primeira versão estável de Go foi lançada em 28 de março de 2012, Go1. A versão Go1 contava com o coletor tradicional, *Mark-Sweep* (JONES; LINS, 1996). Em maio do ano seguinte, lançada a versão Go1.1, compatível com a versão anterior, com melhorias no coletor (do mesmo *Mark-Sweep*) com maior precisão, mas com muitas pausas para coletas. Essas mesmas características se mantiveram na versão Go1.2, em dezembro do mesmo ano.

As primeiras versões de Go eram compostas com o chamado TCMalloc, assim chamado porque ele atribuía cada *thread* ao *cache* local do referido *thread*, com pequenas alocações, onde correriam periódicas operações para migrar a memória de volta a cada *cache* do *thread* local. Porém, essa forma de coleta apresentava baixo desempenho, assim como o número de pausas resultava em muitos momentos de suspensão do programa.

Com o objetivo de reduzir a latência do coletor e permitir maior vazão, foi lançada em junho 2014 a versão Go1.3. Na versão Go1.3, o tempo de pausa para coleta foi reduzido significativamente, com o tempo de parada inferior a 10 milissegundos e o uso do algoritmo *Tricolor* (DIJKSTRA et al., 1978), sendo essa, assim, mais concorrente e incremental baseada na ideia proposta de Dijkstra et al. (1978) (HUDSON, 2015), conforme apresentado na Figura 2.

O novo coletor implementado em Go Go1.3 foi bem aceito pela comunidade, principalmente pelo baixo tempo de pausa, ainda usando o *scanning* completo do *heap* de modo conservador, o processo da coleta se inicia com a fase de marcação dos objetos, onde são dedicados 25% do poder de processamento disponíveis na máquina.

# 4 CONTAGEM DE REFERÊNCIA E BASEADO EM REGIÕES COM DETECÇÃO DE ESTRUTURAS CÍCLICAS

O capítulo anterior apresentou a linguagem de programação Go que usa a técnica de *Mark-Scan* de forma concorrente, com o uso de barreira de escrita para redução do número de pausas, chegando à marca de tempo de pausa inferior a 10 milissegundos. Entretanto, a abordagem utilizada na linguagem Go reduz a ação do coletor de modo que, caso haja um grande número de suspensões, o ciclo de coleta é postergado de modo a liberar menos memória após um ciclo completo. Esse adiamento de coleta aumenta o número de coletas ao longo da execução do programa, podendo levar a um alto impacto em sistemas de tempo real e em sistemas contínuos. Outro problema presente nessa abordagem são as fragmentações internas e externas, onde o objeto alocado não usa todo o espaço de memória disponibilizado.

Pelas circunstâncias acima, introduzimos a coleta por contagem de referência na linguagem Go. Sabe-se que essa técnica executa de modo incremental onde a carga de trabalho é distribuída ao longo da execução do programa, o algoritmo em si não faz uso de pausas, mas, com o problema de coletas de estruturas cíclicas, são utilizadas varreduras locais, o que pode introduzir pequenas pausas, e isso é apresentado mais adiante.

## 4.1 CONTAGEM DE REFERÊNCIA EM GO

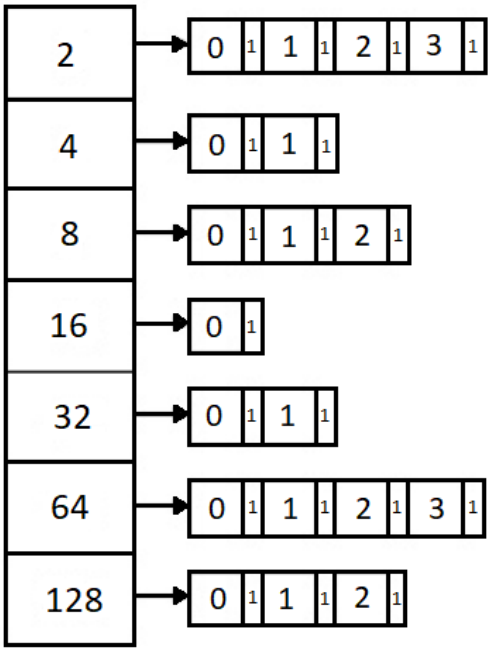
Encontra-se na literatura que a abordagem com contagem de referências é um dos métodos mais incrementais, que interopera diretamente com o programa em execução, evitando os longos períodos de suspensão do programa de usuário causados pelo *stop-the-world*. Para introduzir a contagem de referências em Go, foram inseridas modificações na barreira de escrita emitida pelo compilador bem como uma série de outras modificações, descritas a seguir.

A primeira modificação necessária foi adicionar o contador de referências para cada alocação que escapa da pilha para a *heap*, modificando o alocador dinâmico em Go, ela não dá suporte para desalocações de itens dentro de um único segmento de forma individual, logo, se um objeto dentro de um segmento não é mais usado, o mesmo precisa passar por todo o processo de coleta. Além disso, todo o segmento precisa ser liberado de forma conjunta, ou seja, esse mesmo espaço de memória não poderá ser reutilizado até que todo o ciclo de coleta termine. O espaço de memória somente poderá ser liberado quando todo o bloco no qual está inserido também estiver disponível para ser liberado, mesmo assim, não há garantias de que esse segmento será liberado nesse ciclo de coleta. Com o contador de referências individualizado por objetos, cada elemento dentro de um segmento poderá ser automaticamente reutilizado.

Cada objeto tem um contador de referências e quando o seu valor é decrementado para

zero, seu índice é inserido no topo da lista livre para ser reutilizado, o valor da referência nunca chega a ser igual a zero. Isso permite uma recuperação automática de memória dos objetos que não serão mais utilizados. A Figura 5 ilustra a nova lista livre segregada com os contadores individuais por objeto alocado. As caixas de maior tamanho com valores de 2 a 128 representam os diferentes tamanhos de blocos de memória, ou seja, cada objeto a ser alocado com tamanho igual ou aproximado a eles serão alocados nesse segmento. Em Go são definidos os valores do tamanho de cada lista de maneira estática, ou seja, os valores são predefinidos e os blocos vão sendo liberados à medida que são alocados novos objetos. Para objetos com tamanho de valor é maior que os blocos predefinidos, Go realiza a junção de blocos em um único para alocar o novo objeto.

Figura 5 – Exemplo de lista livre segregada com contador de referências.




**Fonte:** o autor.

Dentro de cada segmento temos os índices que indicam a posição em que os mesmos são encontrados. Para cada bloco, é mantido um índice que está disponível para ser alocado, pode-se observar que todos apresentam o valor de referência inicial igual a um.

O processo de alocação é realizado em duas etapas. Na primeira etapa, todos os blocos estão livres, cada segmento de tamanho fixo tem o índice do primeiro elemento livre, ou seja, a posição zero. À medida que os elementos são alocados nesse segmento, o valor do índice é incrementado e o programa continua sendo executado. Cada posição de memória desse segmento que é liberada fica disponível na lista livre; assim que o índice chegar ao limite das posições livres, todas as alocações futuras para esse segmento retiram a nova posição livre a partir da lista livre encadeada, onde o primeiro elemento sempre será a

última posição que ficou livre.

A segunda etapa é a modificação do compilador de Go para retirar a restrição da barreira de escrita. A remoção permite que seja possível atualizar os contadores dos objetos alocados em *heap*. Toda atribuição ou definição de variáveis e valores que escapam para a *heap* gera uma modificação no contador do objeto alocado. Exemplos:

```
1     var i int = j // i copia o valor de j.

3     var p *int = q // p e q apontam para a mesma ponto.

5     var p *int = &i // p aponta para i.

7     var p *int = new(int) // um novo valor como inteiro onde p aponta para ele.

9     var a [3]int
      var s []int = a[:] // s.ptr aponta para a
```

A passagem de valores entre funções pode causar a alocação do objeto no *heap*. Para cada nova referência a um objeto no *heap*, como ilustrado acima, a barreira de escrita é inserida pelo compilador, e durante as chamadas da barreira de escrita são atualizadas as referências dos ponteiros envolvidos. Esse passo aumenta a atuação da barreira de escrita durante toda a execução do programa, porém essa mudança apenas inclui a operação de atualização de contadores dos objetos envolvidos.

Uma melhoria no processo do uso das barreiras de escrita foi feita a partir dos objetos do *heap*. Observa-se que não são todos os objetos do *heap* que sofrem alteração pela barreira de escrita, logo, esses objetos não precisam de contadores, isso reduz o impacto do uso de memória com o coletor por CR.

## 4.2 DETECÇÃO DE ESTRUTURAS CÍCLICAS CONCORRENTE EM GO

A incapacidade do algoritmo de contagem de referências não coletar objetos que se encontrem em uma estrutura cíclica não é um problema atual, mas já bastante conhecido na literatura (MCBETH, 1963). A primeira solução geral para o problema foi proposta por Martinez, Wachenchauzer e Lins (1990), a qual adiciona um passo a mais no processo, realizando uma varredura em busca de ponteiros que se encontram em ciclo para também serem coletados.

Foram apresentadas melhorias para a proposta da varredura, reduzindo o espaço da busca por ciclos com o adiamento da varredura local. Verificou-se que a maioria dos ponteiros compartilhados não fazem parte de um ciclo, implicando em um desperdício de recursos ao realizar a varredura local, assim Lins (1992a) inseriu o uso de de uma nova file com os ponteiros dos objetos quando os contadores são decrementados para evitar a realização de uma varredura local toda vez que um ponteiro de uma célula com várias referências for excluído.

Em (LINS, 2002) foi o algoritmo de CR com detecção de estruturas cíclicas foi otimizado novamente, evitando uma passagem pelo subgrafo abaixo de um ponteiro excluído compartilhado, a complexidade do espaço também é reduzida como apenas um bit necessário para armazenar as duas marcações de algoritmo. Lins rastreia o lixo compartilhado em vez de dados ativos como o coletor padrão de MS. O algoritmo de Lins, portanto, depende do fato de o compartilhamento ser relativamente raro e de haver poucos efeitos colaterais, já que a rotina de atualização dos ponteiros é mais caro que a versão padrão. Também é demonstrado que o algoritmo com detecção de ciclos é facilmente paralelo a arquiteturas de memória compartilhada Lins (2008).

A versão mais recente do algoritmo para contagem de referências cíclicas é apresentada em Formiga (2011). Essa técnica é uma abordagem híbrida, combinando a contagem de referências com marcação e varredura. Isso pode caracterizar maior complexidade, mas, como apresentado nos artigos supracitados, o adiamento da varredura local implica em grandes benefícios por reduzir a área de busca. Além disso, o contador de cada célula armazena um identificador que caracteriza uma cor, em que todas as cores possíveis são: verde (indica que a célula está em uso); vermelho (mostra que a célula está sendo varrida localmente para verificação de ciclos); e a cor preta (informa qual célula será varrida localmente posteriormente). Esse algoritmo compõe a base da implementação apresentada neste trabalho.

Lins propõe o uso da *status analyser*, uma estrutura de dados auxiliar para guardar ponteiros para todas as células que poderão ser varridas durante a detecção de ciclos. A varredura local somente é iniciada caso não haja mais células livres da lista livre no momento em que é alocado um novo elemento na *heap*. A rotina para obtenção de uma nova célula através da função *new* é mostrada no Algoritmo 10.

**Algoritmo 10:** Alocação de uma nova célula na CR cíclicas.

```
1  Function new() =
2      if free_list is not empty then
3          new_cell = allocate()
4      else
5          if status_analyser is not empty then
6              scan_status_analyser()
7              new_cell = new()
8          else
9              abort "Memory Exhausted"
10     new_cell.color = green
11     return new_cell
```

Quando é alocado um novo objeto, a função busca uma posição livre na lista livre. Assim que uma posição é disponibilizada, o contador inicial é igual a 1 e sua cor é verde;

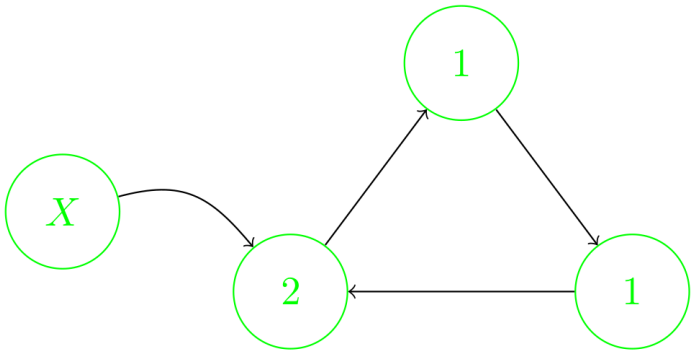
caso não haja nenhuma posição livre, o processo de varredura local se inicia. Neste contexto se encontra uma importante melhoria presente nesse trabalho, dado que usamos o processo de alocamento com listas segregadas, quando não há mais novas células a serem alocadas, ou seja, nenhuma lista com tamanho satisfatório para o objeto a ser alocado foi encontrada. Após isso, é iniciada a varredura local na lista em que o objeto se enquadra, ou seja, o espaço de varredura de busca é menor, pois cada lista livre segmentada dispõe de sua própria *status analyser*, onde é feita a busca entre os objetos alocados neste segmento da memória.

O procedimento de varredura por ciclos é chamado de *scan-status-analyser*, como mostra o Algoritmo 11, ele examina as células presente na estrutura do *status analyser*, que, na atual implementação são os objetos marcados pela cor preta dentro de um segmento de alocações (lista segregada).

**Algoritmo 11:** Rotina de varredura do *status analyser*.

```
1  Function scan_status_analyser() =
2      S = select_from(status_analyser)
3      if S == nil then
4          return
5      if S.color == black then
6          mark_red(S)
7      if S == red and S.rc > 0 then
8          scan_green(S)
9      scan_status_analyser()
10     if S.color == red then
11         collect(S)
```

Figura 6 – Exemplo de situação pré-ciclo.




**Fonte:** Formiga (2011).

Vale ressaltar que todo objeto é considerado um candidato para varredura local, quando seu contador de referências é decrementado para o valor igual a 1. A Figura 6 ilustra esse momento, onde a célula com valor de referência igual a 2 irá perder a referência quando $X$ for desalocado, logo um ciclo se forma, impossibilitando a coleta dos elementos ligados transitivamente. A varredura só é feita em objetos com valor de referência igual a 1 e com a marcação preta. Isso determina que esse objeto é um possível candidato a estar em ciclo, essa marcação da cor preta, ocorre durante a barreira de escrita, que é o momento em que os valores das referências são atualizados, o Algoritmo 12 mostra como é feita a atualização de ponteiros durante a barreira de escrita.

**Algoritmo 12:** Atualização de ponteiros durante a barreira de escrita.

```
1 Function update(R, S) =
2     delete(*R)
3     S.rc = S.rc + 1
4     *R = S
```

O método *delete* significa que a célula será liberada e seu sub-grafo precisa ser alterado. Diferente do algoritmo original de CR, a célula é compartilhada e precisava ser examinada posteriormente para verificação de ciclos. Por isso, a célula é pintada com a cor preta e adicionada ao *status analyser*. O Algoritmo 13 descreve essa rotina. Esse processo também ocorre durante a barreira de escrita.

**Algoritmo 13:** Remoção de ponteiro durante a barreira de escrita.

```
1  Function delete(S) =
2      if S.rc == 1 then
3          for each T in S.children do
4              delete(T)
5          S.color = green
6          link_to_free_list(S)
7      else
8          S.rc = S.rc - 1
9          if S.color ≠ black then
10             S.color = black
11             add_to_status_analyser(S)
```

O primeiro passo ao iniciar a varredura é a realização da operação de marcação vermelha, na qual são percorridas as células compartilhadas para detectar ciclos como mostra o Algoritmo 14. Ao iniciar em uma célula, a mesma é marcada da cor preta para vermelha em seguida o contador de todas as células que apontam para essa célula são decrementados, de modo a remover as referências internas a um ciclo ou sub-grafo local (FORMIGA,

2011). Todas as células decrementadas são visitadas de forma recursiva pela operação de marcação vermelha. Com isso, todo o sub-grafo a partir da célula de entrada é visitado, caso nenhuma célula no sub-grafo possua um contador com valor maior que zero, isso caracteriza um ciclo e o mesmo pode ser coletado.

**Algoritmo 14:** Rotina para marcação de células de vermelho.

```
1  Function mark_red(S) =
2      if S.color ≠ red then
3          S.color = red
4          for each T in S.children do
5              T.rc = T.rc - 1
6          for each T in S.children do
7              if T.color ≠ red then
8                  mark_red(T)
9              if T.rc > 0 and T is not in status_analyser then
10                 add_to_status_analyser(T)
```

Porém, quando é encontrada uma célula que possui o contador maior que zero, isso indica que há ao menos uma referência externa ao sub-grafo e todas as células no fechamento transitivo podem ser consideradas ativas, pois as mesmas ainda são alcançáveis a partir da raiz do programa. Quando não é identificado um ciclo, é necessário restabelecer o estado anterior à varredura local. O procedimento de marcação verde é responsável por percorrer as células ligadas, isso é apresentado no Algoritmo 15, marcando a célula de verde e incrementando o valor de referência em 1, isso também é feito de forma recursiva, desfazendo toda a marcação feita. No final de toda a varredura de marcação, todo objeto do segmento marcado de vermelho pode ser apontado pela lista livre.

**Algoritmo 15:** Rotina para varrer células em verde.

```
1  Function scan_green(S) =
2      S.color = green
3      for each T in S.children do
4          T.rc = T.rc + 1
5          if T.color ≠ green then
6              scan_green(T)
```

Durante o processo, onde todas as células são pintadas primeiramente de vermelho e logo após de verde, se nenhuma célula no sub-grafo possuir um contador com o valor maior que zero, isso indica que não há evidências de que o mesmo está transitivamente conectado a alguma raiz e pode ser coletado. Nessa rotina, deve-se remover toda a referência de $S$ para todas as células apontadas por ela, para que $S$ seja colocada na lista livre novamente.

Caso tenha uma célula apontada para $S$ e a mesma tiver cor vermelha, ela também pode ser coletada similarmente (ver Algoritmo 16).

**Algoritmo 16:** Coleta de células inativas após varredura local.

```
1 Function collect(S) =
2     for each T in S.children do
3         delete(T)
4         if T.color == red then
5             collect(T)
6     S.color = green
7     add_to_free_list(S)
```

A principal melhoria apresentada para esse processo de coleta de dados em ciclos, apresentada nesta tese, também diz respeito ao modo como a disposição das alocações é feita, como dito anteriormente, cada bloco de segmento de memória é criado de modo dinâmico durante a execução do programa, de acordo com a necessidade. A varredura local somente é realizada quando não há mais memória disponível. A proposta de Formiga (2011) realiza a varredura em toda o *heap*, por outro lado, nós propomos a redução dessa varredura para um único segmento, o que não mais apresenta espaços livres, isso reduz significativamente o tempo de varredura local. A arquitetura também difere de Formiga (2011) principalmente em como o coletor e mutador interferem um no processo do outro, para isso foi usada sincronização explícita, utilizando travas como proposto por Lins (1992b). Isso é usado na rotina de coleta apenas, para não enfraquecer as asserções da CR.

Cada segmento da lista segregada pode disparar uma varredura local de forma independente quando se encontra sem posições livres disponíveis. A varredura somente será realizada em células que satisfaçam a condição.

O número de varreduras em um segmento de memória é dado pelo número de objetos presentes naquele segmento no máximo, tornando este ponteiro um candidato à varredura local. Outro detalhe sobre a implementação do trabalho apresentado está relacionado à abordagem preguiçosa, ou seja, caso seja encontrado um ciclo na primeira varredura realizada do segmento, a mesma é retornada e o programa continua sua execução.

## 4.3 CONTAGEM DE REFERÊNCIA CÍCLICA BASEADA EM REGIÕES DE MEMÓRIA

Nesta seção, propomos uma nova abordagem utilizando contagem de referências com detecção de ciclos, gerenciamento de memória e coleta por região de memória.

Jones e Lins (1996) evidenciam empiricamente que a maioria das células não passam de referência igual a um (RC=1), além disso, possuem um curto tempo de duração de vida. Apesar das otimizações e melhorias apresentadas na seção anterior, reduzindo sig-

nificativamente o tempo de suspensão do programa, a abordagem ainda pode apresentar pausas quando há bastantes ciclos a serem coletados.

A Figura 7 mostra o exemplo de como o algoritmo de CR funciona para realizar a varredura local, mostrando a diferença na detecção de ciclos ao reduzir o espaço da varredura por blocos de memória.

A situação inicial mostra um ciclo (A, B e C), que apresenta um único ponto transitivamente conectado ao conjunto raiz, sendo esse o objeto D, em seguida D é liberado, marcando o objeto B como candidato a ser varrido, porém a lista segregada livre em que B está alocado ainda tem espaços livres, logo os objetos F e G podem ser alocados.

A varredura só é iniciada quando um novo objeto precisa ser alocado e esse objeto tem tamanho equivalente a lista livre do objeto B, logo a varredura é iniciada a partir de B. Após a rotina de marcação vermelha e se identifica que o ciclo foi desconectado do conjunto raiz caracterizando o ciclo como inativo o mesmo pode ser coletado pela rotina de *collect*, devolvendo as células para a lista livre.

A varredura local atua somente na lista de objetos com tamanho fixo em que não há mais espaços livres para uma nova alocação, reduzindo o tempo de pausa para coleta de ciclos, sendo essa uma das diferenças aqui presente que diferenciam da proposta por Formiga (2011). Todo o processo de marcação presente em Formiga (2011) usa um bit para cada objeto na *heap*, isso aumenta o consumo total de memória utilizada, aqui usamos a marcação de bits, que é uma estrutura auxiliar criada somente no momento da execução, esse processo de marcação será explicado a seguir.

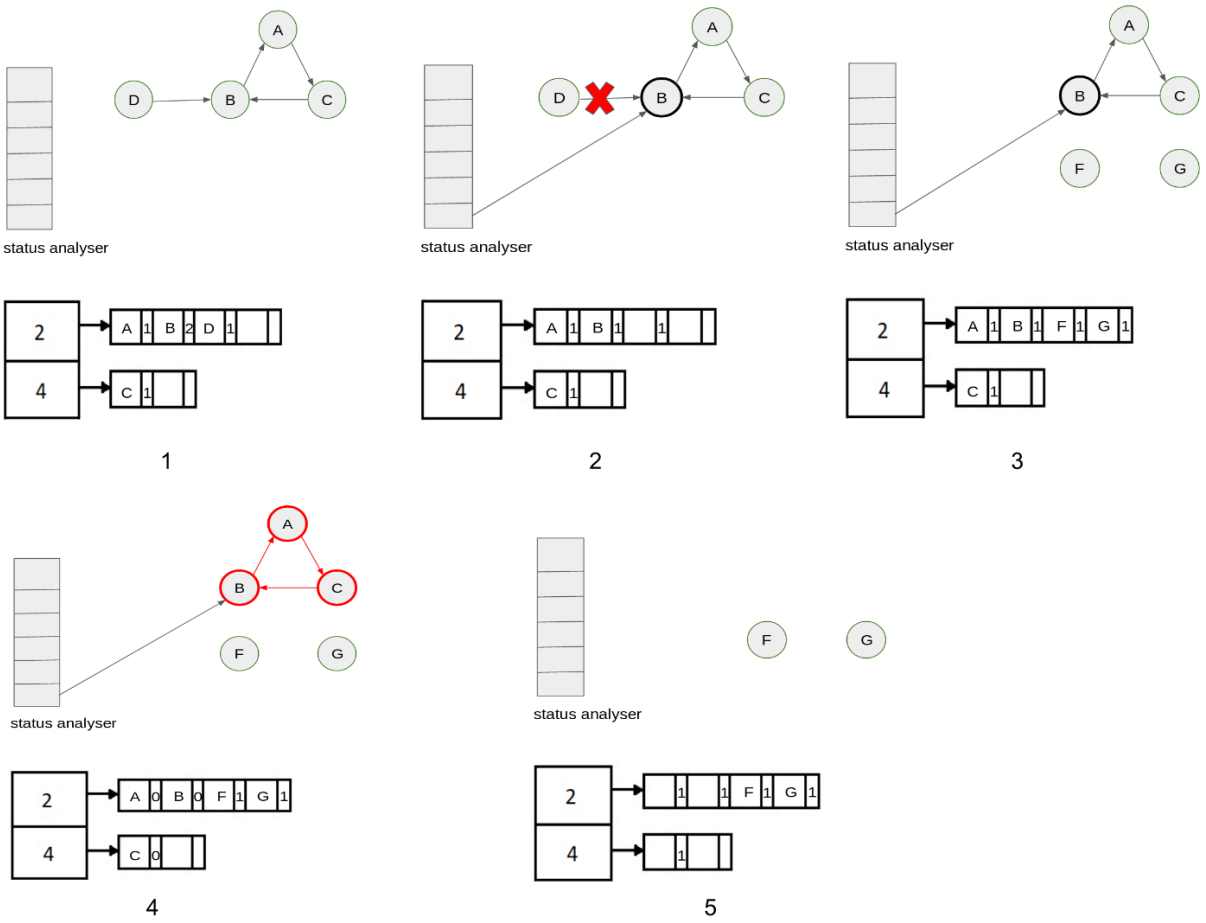
### 4.3.1 Marcação de *bits*

Além do problema de coleta de objetos dispostos de maneira circular, a contagem de referências apresenta mais duas principais problemáticas: a sobrecarga de trabalho para atualizar os contadores e a necessidade de maior espaço físico para armazenamento dos contadores por objeto no *heap*.

Uma das melhorias proposta, está na redução do custo de sobrecarga de trabalho e espaço em memória. O objetivo é reduzir a quantidade de objetos alocados gerenciáveis pelos contadores de modo a não comprometer o funcionamento do algoritmo. Foi observado que variáveis globais e outras acabam por escapar da pilha para a *heap*, isso também se dá porque alguns objetos alocados são apontados por algum ponteiro (podendo ser até mesmo nulo) diretamente da pilha. Isso gera referências inativas, que acabam por não apresentar mudanças durante toda e execução do programa. Esses objetos estariam desperdiçando espaço de memória, em alguns casos, incluindo barreiras de escrita de modo a sobrecarregar o sistema.

Além disso, para pequenas alocações, como pequenas *strings* e outras alocações consideradas mínimas, não são adicionados contadores de referências. Esses objetos também

Figura 7 – Ciclo inativo com varredura local por lista livre segregada.




**Fonte:** o autor.

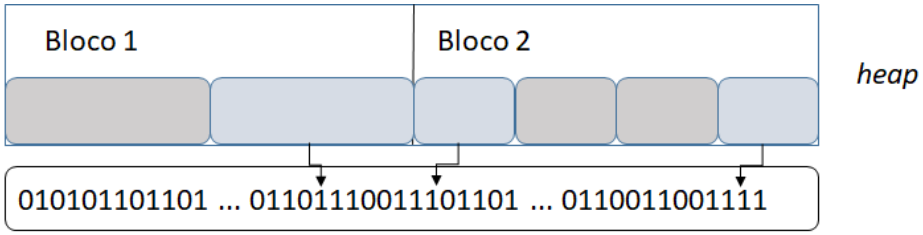
são alocados em um segmento único de memória, onde esses blocos de memória resultantes são liberados apenas quando todos os objetos estão inacessíveis.

A solução apresentada por Formiga (2011) é eficaz na coleta, mas exige uma quantidade maior de memória, além disso, o espaço necessário para conter os contadores por objeto adiciona mais uma exigência de espaço para marcação das cores por objeto. Para minimizar o consumo de memória, propomos o uso de marcação de *bits* (*markBits*) e *bitmap*, abordagens que fornecem acesso ao bit marcação de um objeto na *heap*, reduzindo a necessidade de maior espaço de alocamento. Utilizando essas técnicas foi, observado um aproveitamento de até 20% da *heap*.

A marcação de bits é muito utilizada para coletores com base na marcação e varredura (GARNER; BLACKBURN; FRAMPTON, 2007), onde cada bit representa uma palavra ou um objeto na *heap*. A coleta de dados em ciclos é uma abordagem similar, a impacta na sobrecarga de memória causada pela marcação das células durante a varredura local.

Durante a varredura, é preparado um novo *bitmap* de marcação para a próxima fase de marcação, como ilustrado na Figura 8. Ele percorrerá a lista de extensões em uso e usará

Figura 8 – Exemplo de marcação de bits.




**Fonte:** o autor.

um alocador linear de estilo de arena simples para atribuir a cada extensão um *bitmap* de marca dimensionado para o número de objetos nesse segmento.

A nova abordagem é proposta com base em resultados relevantes com alto desempenho e baixo uso de memória em Davis (2015), utilizando a abordagem de gerenciamento de memória baseado em regiões (GROSSMAN et al., 2002).

### 4.3.2 Gerenciamento de memória baseado em regiões

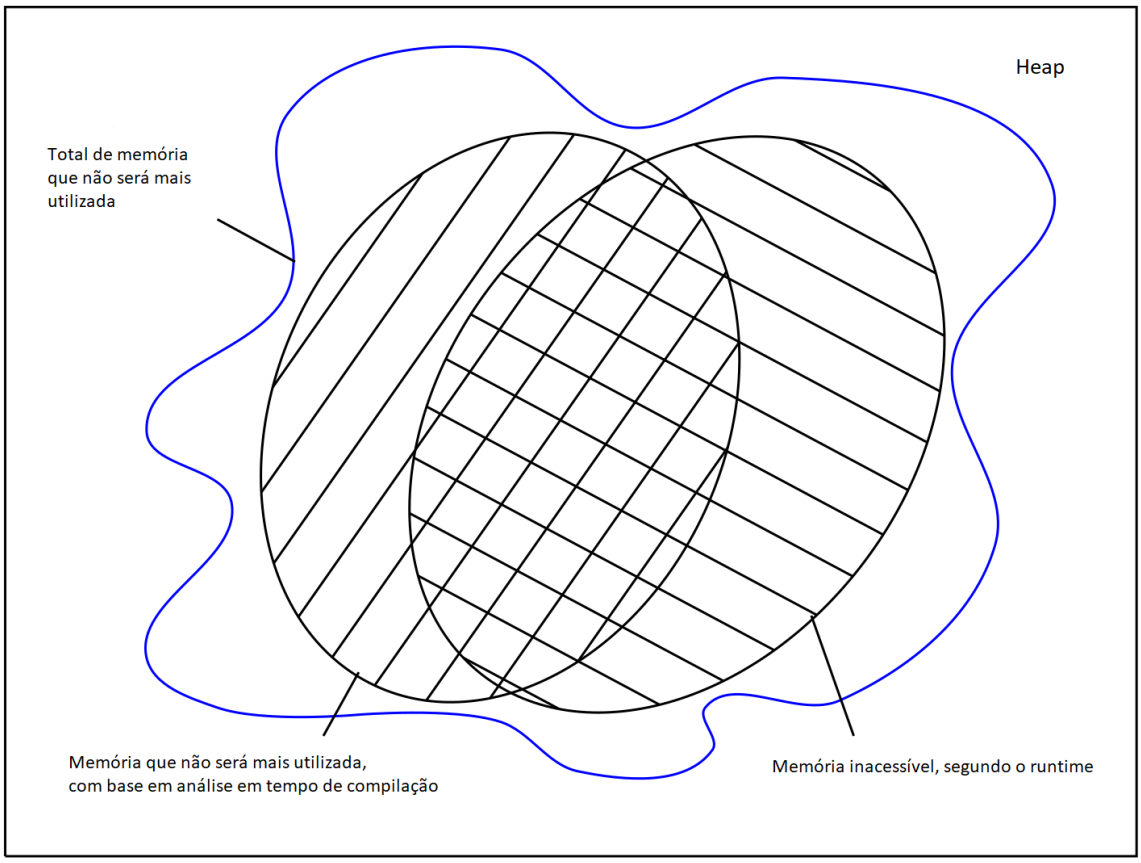
Gerenciamento de memória baseado em regiões trata da combinação de análise estática e a instrumentação do código no tempo de execução. Essa abordagem foi primeiramente observada por Tofte e Talpin em 1994 e 1997, essa técnica foi implementada na linguagem por Cyclone Grossman et al. (2002).

Dada a inferência, ou seja, posição do código durante a compilação é possível estimar a proximidade do uso de objetos, com isso podemos determinar quais objetos podem ser agrupados para serem coletados coletivamente. Assim os mesmos podem ser alocados de modo agrupado em uma única região de memória (DAVIS, 2015). Como ilustrado na Figura 9, essas técnicas são responsáveis pela coleta de dados em áreas diferentes, não havendo sobreposição em nenhum caso, unindo o melhor das duas técnicas. Essas técnicas podem eliminar as desvantagens presentes em cada uma de forma individual, permitindo um alto desempenho e reúso de memória.

Davis (2015) modificou o compilador Gccgo de Go, substituindo as alocações da *heap* para alocações por regiões limitadas, garantindo a transformação de código por verificação formal que as transformações realizadas não apresentam efeitos colaterais ou alterações no funcionamento do programa escrito em Go. Sendo o coletor baseado em regiões de memória, uma abordagem principalmente estruturada na análise e transformação de código durante o tempo de compilação. Isso reduz as ações necessárias durante o tempo de execução, permitindo que a contagem de referências com detecção de ciclos fique responsável por uma região menor (a depender do programa), de modo que a possibilidade de encontrar ciclos nessa região são maiores.

A proposta apresentada por Davis (2015) mostra os benefícios de alocações por região de memória, destacando que nem sempre é possível designar o tempo de vida de um objeto.

Figura 9 – Uso de memória: visão estática vs dinâmica.




**Fonte:** Davis (2015).

Sendo assim, a região com objetos remanescentes podem ter um tamanho equivalente a quase todo o *heap*. Além disso, os objetos que são alocados nessa região global são objetos com forte acoplamento e dependências, podendo concluir que tais objetos podem se caracterizar como potenciais objetos em ciclo.

Go é uma linguagem que ainda continua em desenvolvimento. Novas versões são lançadas periodicamente, levando a uma atualização do compilador Gccgo usado por Davis (2015) da versão 4.6.3 para a versão 4.7.4. No código original, é feita a coleta por cópia para a região global, composta pelos objetos em que o tempo de vida não eram passíveis de serem estimados. Nesta abordagem, foi utilizada a contagem de referências com detecção de ciclos, onde o tamanho da página de região de memória é fixa em 4.096 *bytes*, tamanho padrão para sistemas com 64-*bits* no Linux.

A coleta por regiões de memória feita por Davis (2015) não alterou diretamente o código fonte do Gccgo, pois foi construído como um *plugin* onde as chamadas aos métodos nativos de *malloc* são substituídas por chamadas aos métodos de *createRegion*.

Toda a manipulação de código permite que objetos com tempo de vida semelhantes sejam alocados juntos, agrupando objetos sem ciclo, e objetos que estão transitivamente ligados de forma dependente também são alocados na mesma região, reduzindo, assim, a região em que o contagem de referências é feita. Além disso, essa abordagem retira a necessidade de maior espaço em memória para alocar o contador de referência de forma individual. Não foi preciso alterar a análise e manipulação de código feita em Davis (2015); o autor proporcionou o acesso ao código fonte utilizado em sua pesquisa.

## 4.4 TRABALHOS RELACIONADOS

O gerenciamento de memória é uma área de pesquisa ativa com diversos estudos e relatos. Também é de conhecimento acadêmico que esta área não se caracteriza como um assunto trivial, permanecendo relevante mesmo com o decorrer dos anos.

Diversas abordagens foram apresentadas ao longo dos anos, combinando soluções como contagem de referência com barreiras de sincronização (AZATCHI; PETRANK, 2003), enquanto outras abordagens focam no uso da coleta baseada no serviço, ou seja, baseada no sistema em que as coletas são realizadas quando o sistema não está realizando atividades substanciais (DEGENBAEV et al., 2016). Muitas das pesquisas realizadas denotam melhorias para específicos campos, como sistemas de tempo real (REN et al., 2018) ou sistemas paralelos (LI; WU; CHEN, 2018).

Choi, Shull e Torrellas (2018) apresentam uma contagem de referências tendenciosa, retirando operações atômicas para atualização dos contadores. Isso é feito porque em um sistema *multithread*, uma única *thread* mantém a trava, impossibilitando que os demais contadores sejam atualizados, assim, gerando um gargalo na execução da coleta, porém, para tal, abordagem é necessário manter mais de um contador por objeto por *thread* (CHOI; SHULL; TORRELLAS, 2018), o que aumenta o custo imposto pela contagem de referências de manter contadores junto aos objetos alocados.

Terei e Levy (2015) apresentam uma Application Programming Interface (API) escrita em Go, que trabalha entre a aplicação e a coleta, chamada de BLADE. Essa API é usada como uma interface que permite ao desenvolvedor escolher o melhor momento para a coleta dos dados da aplicação, permitindo de forma explícita que ocorra o tempo de suspensão da aplicação sem maiores comprometimentos ao serviço (TEREI; LEVY, 2015).

Linguagens com coleta de memória de forma automática são atrativas ao desenvolvedor por não precisarem de acoplamento do gerenciamento de memória junto ao comportamento do sistema. Outro trabalho feito em Go para o aprimoramento da coleta e redução do tempo de pausa foi apresentado por Davis (2015), uma abordagem de gerenciamento de memória com base em alocamento por regiões de memória. Essa técnica realiza análise estática e transformação de código para definir regiões onde os objetos serão alocados e desalocados conjuntamente. Mais informações e artigos relacionados a Go podem ser encontrados em Golang (2019).

A transformação do código permite que o compilador possa estimar o tempo de vida dos objetos que serão alocados, então todos os objetos com tempo de vida similar são agrupados e alocados em uma única região. Assim, os objetos são coletados simultaneamente e toda a região fica livre para a nova alocação. Isso reduz drasticamente o tempo de pausas bem como o estaço total de memória. Esse tipo de abordagem foi proposta na linguagem Cyclone por Grossman et al. (2002). Ao invés de liberar objetos de forma individual, colocando-os na lista livre, o gerenciamento baseado em regiões de memória pode fazer o mesmo. Isso permite a liberação de todo um segmento de memória e não somente um único objeto, a técnica também tira proveito da localidade de *cache*.

O uso de uma solução híbrida utilizando a contagem de referências e coleta por regiões de memória não é inovadora. Gay e Aiken (1998), propuseram o uso de contagem de referências global por regiões.

Uma das contribuições proposta nesta tese faz uso das duas técnicas combinadas: contagem de referências e o gerenciamento de memória baseado em região de memória. Para o gerenciamento de memória de regiões diferentes, sendo coletas separadas, maiores detalhes serão apresentados no Capítulo 4.

O gerenciamento de memória com base em regiões de memória também apresenta desvantagens, desde que os objetos somente sejam liberados quando toda a região de memória puder ser reciclada, implicando que objetos não são liberados automaticamente. Isso também induz que toda a região de memória pode prolongar seu tempo na *heap* esperando que um único objeto esteja apto a ser desalocado, ou seja, um único objeto pode reter toda a região de memória a ser reciclada.

Outra limitação se dá devido ao fato de essa técnica ser baseada em análise estática, em que os objetos são transformados de modo a tentar prever o tempo de vida dos objetos alocados. Porém, sabe-se que essa predição é indecidível (RICE, 1953), logo, quando os objetos não apresentam tempo de vida similar para serem alocados juntos, os mesmos são alocados em uma região única que pode ficar grande devido à quantidade de objetos. Para esses objetos, outro tipo de coleta deve ser usado, podendo causar *memory leaks*.

Neste capítulo foram consideradas as principais técnicas de gerenciamento de memória automático, bem como seus pontos fortes e fracos. Outras abordagens são encontradas na literatura, porém, as técnicas aqui citadas são a base de todos os demais ramos encontrados pesquisados.

# 5 EXPERIMENTOS E RESULTADOS

Go é uma linguagem que pode ser inserida em diversos tipos de cenários e aplicações. Neste trabalho foram usados diversos programas de teste para avaliar o desempenho dos coletores propostos, bem como a carga de trabalho imposta ao sistema.

Para analisar o desempenho das técnicas e algoritmos do Capítulo 4, foi utilizado um conjunto de *benchmarks*, comparando os coletores com MS e CR nos compiladores de Go.

Utilizamos os *benchmarks* presentes no Rosseta Code (ROSETTA, 2019), programas de diferentes categorias, escritos em mais de 700 linguagens diferentes. A maioria dos programas apresenta descrição dos problemas resolvidos, permitindo que seja feita uma análise de desempenho ou usabilidade das linguagens presentes, como em Nanz e Furia (2014).

Foram executados mais de 900 programas escritos por diversos desenvolvedores, envolvendo vários programas, aplicações *Web*, dentre outros. Além disso, os programas que não apresentaram nenhuma atividade de coleta de dados foram descartados.

Fulgham e Gouy (2009) apresentam uma suíte de *benchmarks* chamada *The Computer Language Benchmark Game*. Esse repositório contém diversos programas que buscam avaliar diferentes aspectos do funcionamento das linguagens de programação mais utilizadas. O repositório recebe códigos das mais diversas linguagens onde são propostas entradas de dados cuja saída é padronizada. A execução desses programas é feita em uma mesma máquina de servidor com o sistema operacional Debian, cada resultado é apresentado como um *ranking* na mesma página, sendo esses *benchmarks* utilizados em trabalhos com intuito de avaliar características das linguagens.

Alguns dos programas finalizaram a execução sem que o coletor tenha desempenhado qualquer atividade, principalmente no coletor com base no MS, impossibilitando a comparação das abordagens. Neste trabalho nós utilizamos um critério para seleção dos *benchmarks*, sendo selecionados os *benchmarks* que possuam um consumo de memória de maneira a disparar o processo de coleta.

Da suíte *The Computer Language Benchmark Game* somente o *binary-tree* apresentou dados compatíveis ao definido anteriormente para a análise de consumo de memória e performance da coleta de dados. Além disso, somente o *benchmark matmal*, retirado de Davis et al. (2012), apresentou características semelhantes.

Nesta tese, foram também utilizados outros *benchmarks* utilizados pelo time de desenvolvimento da linguagem Go, envolvendo testes com características específicas para estressar o sistema de coleta de modo a avaliar os aspectos do coletor. A Tabela 2 exibe os *benchmarks* utilizados neste trabalho e suas características.

Tabela 2 – *Benchmarks*

| **Nome do *benchmark*** | Descrição |
|---|---|
| *Benchmark* de pausas | Criado por Sewell (2019) para avaliar o tempo de pausa causado pelos coletores das linguagens, este *benchmark* realiza repetidas trocas de mensagens em um *buffer* de tamanho limitado. As mensagens mais antigas dentro do *buffer* são coletadas de forma contínua e o tamanho da *heap* se mantém cheio durante toda a execução do programa, forçando o coletor a atravessar todo o *heap* para detectar quais objetos ainda possuem referências ativas. A coleta é proporcional ao número de objetos/ponteiros ativos (SEWELL, 2019). |
| Binary-tree | Esse *benchmark* é baseado em operações que realizam o balanceamento de uma árvore binária. A depender das entradas, o número de nós e graus da estrutura da árvore é aumentado, forçando o coletor a liberar mais espaço de memória a cada nível da árvore. (FULGHAM; GOUY, 2009) |
| Matmul | Executa multiplicações de matrizes com complexidade de tempo $O(n^3)$ (DAVIS, 2015). |
| Garbage | Criado para avaliar o próprio coletor da linguagem Go, estressando o coletor. Realiza o parse do pacote *Web net/http* de forma contínua, usando o pacote nativo de Go (go/parser) e descarta os resultados ao final de cada repetição. |
| SmallHeap | Testa o rendimento do coletor em Go com pequenos objetos e alta taxa de alocação. |
| LargeBSS | Utilizado para avaliar o comportamento do coletor da linguagem Go com pequenas pilhas e alta taxa de alocação. |

## 5.1 CONFIGURAÇÃO

Os experimentos foram executados em três servidores diferentes com sistema operacional Ubuntu Server 16.04 LTS (Xenial Xerus) 64-bit. Cada servidor apresenta tamanhos diferentes de memória Random Access Memory (RAM), sendo: 4Gigabyte ($GB$), 8$GB$ e 16$GB$, chamadas de Máquina 1, 2 e 3, respectivamente. Todos os experimentos foram executados de forma isolada, de maneira que não foi feito uso de interface gráfica e nenhum outro sistema além do SO concorre com os testes. Todas as máquinas têm arquitetura com multicores, as configurações das máquinas são descritas na Tabela 3.

Tabela 3 – *Arquiteturas utilizadas*

| **Nome da Máquina** | **Processador** | **Memória Principal** |
|---|---|---|
| Máquina 1 | AMD Phenom II X6 1090T 3,2GHz | 4 *GB* |
| Máquina 2 | Intel i7-3770 3,4Ghz | 8 *GB* |
| Máquina 3 | Intel Core i7-8550U 1,80Ghz | 16 *GB* |

*Os experimentos foram executados com o sistema operacional Ubuntu Server 16.04 LTS (Xenial Xerus) 64-bit

## 5.2 AMBIENTE DE EXECUÇÃO DOCKER

Durante o decorrer dos experimentos, observamos que os trabalho consultados não apresentam, de forma clara, os procedimentos para reprodução dos experimentos ou arquivos fontes, tornando inviável a reprodução dos experimentos, impossibilitando a comparação ou análise de algumas abordagens encontradas. Desse modo, visando à possibilidade de réplica e fácil acesso ao ambiente e configuração descrita nesta tese, usamos a API de microsserviço chamado Docker (DOCKER-HUB, 2019). Essa API provê a reprodução dos experimentos e adoção das técnicas apresentadas ao longo do trabalho.

Docker é uma plataforma que opera no nível do sistema operacional de modo virtualizado, também conhecido como *containerization* (RAD; BHATTI; AHMADI, 2017), atuando por camadas de virtualização. Além disso, Docker permite que se tire total proveito da máquina hospedeira, facilitando a reprodução de ambientes computacionais. Essa ferramenta tem sido adotada na indústria e pesquisas científicas (BOETTIGER, 2015).

Rad, Bhatti e Ahmadi (2017) mostram que a utilização de Docker não degrada o desempenho da máquina devido à virtualização. Além disso, possui suporte para diversas plataformas, tais como: Linux e Windows. Essa ferramenta é principalmente utilizada para o desenvolvimento e ambiente de serviços *Web*.

A imagem binária de Docker pode conter todos os softwares e suas configurações. Diferentemente dos demais serviços de virtualização, é possível usar os recursos totais da máquina hospedeira de forma otimizada. A imagem com todo o processo de instalação e configuração está disponível para uso no Apêndice B desta tese ou para download através do link: https://gitlab.com/filipevarjao/go-rbmm-refcount/. A imagem disponível já contém a versão do Gccgo 4.7.4, com as alterações do Go *front-end* e o *plugin* de gerenciamento de memória com base em regiões.

Todas as alterações presentes feitas pelo coletor de Go tiveram como base a versão 1.9. Encontram-se disponíveis as versões com contagem de referências e detecção de ciclos no link: https://github.com/filipevarjao/go.

## 5.3 ANÁLISE ESTATÍSTICA

Para esclarecer a ambiguidade nos resultados, foi realizado o teste estatístico $U$ de *Mann-Whitney* (KLOTZ, 1966) para os coletores em Go com MS e CR. O teste estatístico $U$ de *Mann-Whitney* (KLOTZ, 1966), também conhecido como teste da soma dos postos de *Wilcoxon*, *Wilcoxon-Mann-Whitney* ou teste de *Mann-Whitney* é baseado nos postos dos valores obtidos, combinando-se as duas amostras, sendo esse teste não-paramétrico (não assume uma distribuição normal).

A hipótese nula, representada por $H_0$, determina que as médias em análise são iguais. No teste de hipótese, é avaliada a probabilidade de as amostras terem o mesmo valor do parâmetro real. A hipótese oposta é chamada de hipótese alternativa, representada por $H_a$ e ela contradiz a hipótese nula. Caso a hipótese nula seja rejeitada, a hipótese alternativa é válida. (MCKNIGHT; NAJAB, 2010).

O teste estatístico é realizado a partir dos dados das amostras e depende das características do problema. Além disso, o teste estatístico produz uma informação de probabilidade condicional chamada de $p-value$, sendo este a probabilidade de se obter uma estatística de teste mais extrema do que aquela observada em uma amostra, assumindo verdadeira a hipótese nula. Quanto menor o valor do $p-value$, mais evidência se tem contra a hipótese nula.

Para esses resultados em que a diferença dos valores não é presente, foi utilizada a validação cruzada com 10-*fold* 30 vezes para cada um dos coletores. Assim, foram calculadas as médias para cada *fold*, totalizando 300 execuções para os *benchmarks* e para todas as entradas em cada arquitetura.

## 5.4 RESULTADOS

Os *benchmarks* são de naturezas diferentes, porém, todos apresentam coletas, alguns têm entradas diferentes de dados, logo, para os testes com variação na entrada, foram produzidos gráficos para ilustrar o desempenho.

Os demais *benchmarks* com execuções únicas são apresentados em tabelas de dados; é perceptível que, por muitas vezes, os resultados são similares, sendo imperceptíveis as diferenças dos dados nos gráficos, assim são apresentados os valores em tabelas.

Os testes apresentados a seguir representam a coleta em Go com o algoritmo concorrente *Tricolor* MS com ambos os compiladores, Gogc e Gccgo. Utilizamos a coleta por contagem de referência no compilador nativo de Go, enquanto no Gccgo foi usada a abordagem com base em regiões de memória e contagem de referência. Para todos os experimentos foram removidas as operações de *IO*, para que não houvesse perda no tempo de execução para impressão de dados.

Os valores referentes à memória coletada são calculados a partir do número de objetos desalocados, logo, não se tem o valor exato da memória desalocada. Em contrapartida,

temos o número de objetos desalocados durante a execução dos experimentos, permitindo estimar os valores de memória.

As variáveis que serão apresentadas das tabelas e figuras discorrem sobre os valores em média do tempo de execução do programa, média do total de memória alocada durante toda a execução, o número médio de pausas realizadas, pior tempo de pausa e valor aproximado da memória liberada pelos coletores. Além disso, os valores do compilador Go com coleta por MS são representados por `go_ms`. Enquanto com o mesmo sistema de coleta desenvolvido com compilador do GCC é dado por `gcc_ms`, `go_cr_std` apresenta os resultados para compilador Go com coleta por CR e detecção de ciclos originais desenvolvidos por Formiga (2011); enquanto `go_cr` apresenta o algoritmo de CR com as melhorias e otimizações propostas nesta tese, a junção da abordagem de CR e gerenciamento baseado em regiões foi unicamente usado com o compilador GCC o qual é apresentado como `gcc_crbr`.

### 5.4.1 *Benchmark* de Pausas

Um dos aspectos apresentados a seguir é o tempo de pausa, como descrito na Tabela 2. O *benchmark* de pausas realiza um milhão de trocas de mensagens em um único *array* de *buffer* com tamanho fixo, forçando a aplicação a fazer novas coletas para que haja espaço suficiente para a nova alocação.

O teste realiza um *loop* com 1 milhão de repetições, em busca do pior tempo de envio das mensagens. Os testes apresentados foram obtidos a partir da média de 30 execuções para cada coletor, nas diferentes máquinas. Esse envio de mensagem é dado por uma passagem de valores, podendo ser o gatilho para a coleta de dados.

O mesmo *benchmark* proposto por Sewell (2019) foi usado para Go, OCaml, HaskellGHC, Java e outras. A Tabela 4 mostra os resultados para as máquinas 1, 2 e 3. Também são computados os valores totais de execução do programa, total de memória alocada e o número de pausas realizadas por cada coletor.

Tabela 4 – Resultados para o *benchmark* de pausas.

| **Coletor** | **Tempo de execução** | **Pior tempo de pausa** | **Total de mem. alocada** | **Nº de pausas** |
|---|---|---|---|---|
| **Máquina 1** | | | | |
| go_ms | 1,54$s$ | 18920$us$ | 1,02$GB$ | 11 |
| gcc_ms | 3,40$s$ | 47290$us$ | 1,05$GB$ | 11 |
| go_cr std | 1,9$s$ | 815,2$us$ | 1,53$GB$ | 0 |
| go_cr | 1,43$s$ | 630,45$us$ | 1,02$GB$ | 0 |
| gcc_crbr | 3,37$s$ | 68,03$us$ | 24,77$KB$ | 0 |
| **Máquina 2** | | | | |
| go_ms | 668,48$ms$ | 6220$us$ | 1,02$GB$ | 11 |
| gcc_ms | 269000$ms$ | 22870$us$ | 1,05$GB$ | 11 |
| go_cr std | 838,5$ms$ | 368,2$us$ | 1,72$GB$ | 0 |
| go_cr | 645,66$ms$ | 282,93$us$ | 1,15$GB$ | 0 |
| gcc_crbr | 295000$ms$ | 21,97$us$ | 24,76$KB$ | 0 |
| **Máquina 3** | | | | |
| go_ms | 705,97$ms$ | 5220$us$ | 1,02$GB$ | 11 |
| gcc_ms | 279000$ms$ | 19990$us$ | 1,05$GB$ | 11 |
| go_cr std | 910,6$ms$ | 387,1$us$ | 1,56$GB$ | 0 |
| go_cr | 702,63$ms$ | 297,09$us$ | 1,02$GB$ | 0 |
| gcc_crbr | 311000$ms$ | 106,93$us$ | 22,44$KB$ | 0 |

É possível verificar que o desempenho do programa de teste de pausas muda à medida que é executado em arquiteturas diferentes, principalmente no tempo de execução e pior tempo de pausa. Os valores de memória alocados e número de pausas são relativamente constantes ao longo das execuções.

Pode-se observar que o limite de espaço total na RAM da Máquina 1 repercutiu no tempo de execução do programa. Também observa-se que a contagem de referências não apresentou suspensão completa ao sistema, onde o pior tempo de pausa teve um número semelhante à execução do programa sem nenhuma coleta.

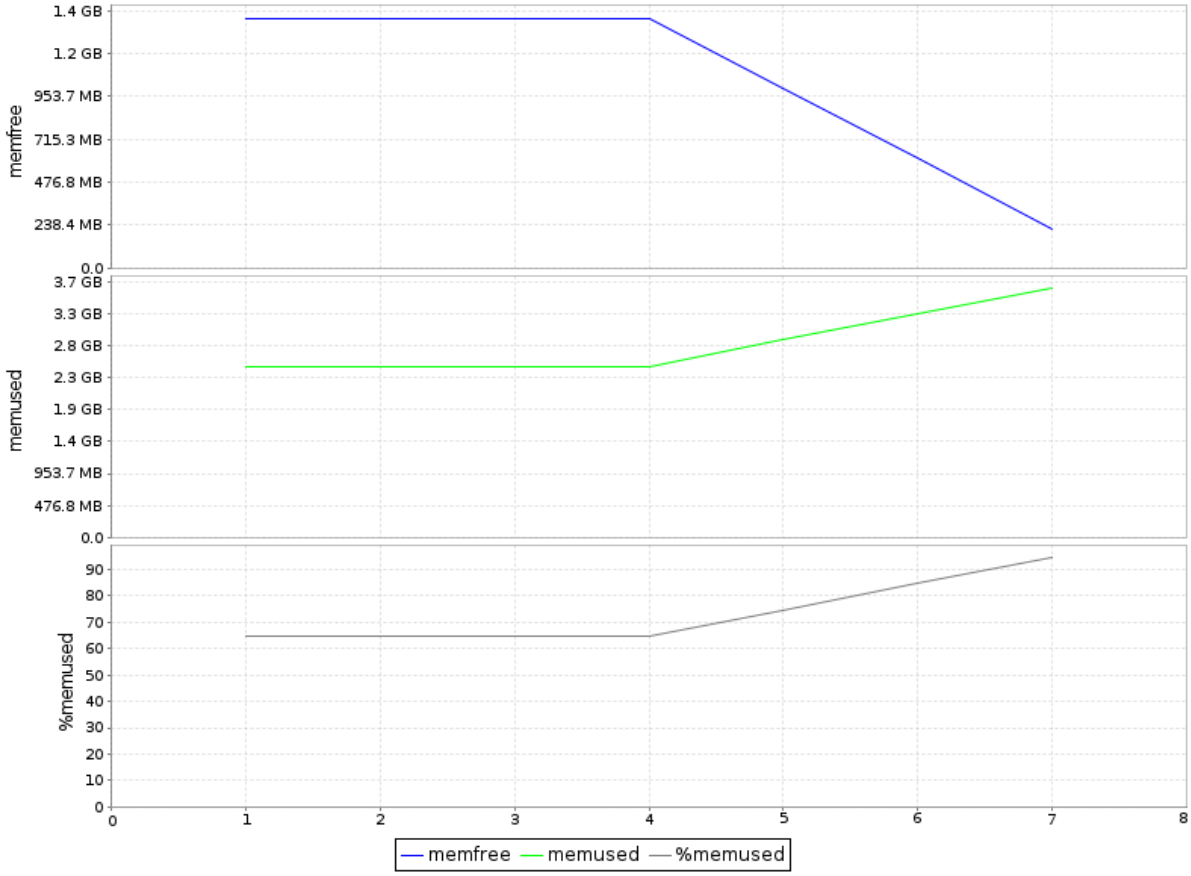
O valor de memória alocada foi similar para a maioria dos coletores, exceto para o coletor com base em regiões de memória. O resultado do coletor por contagem de referências mostra que a alocação com lista segregada tira proveito da fragmentação interna para manter o contador de referências sem o alto custo de requerer mais espaço. Isso é agregado ao fato de não ser necessário manter o contador para todos os objetos no *heap*. Além disso, a coleta por CR é equiparada à coleta MS, porém, apresenta um valor de pausa inferior, impactando diretamente no tempo total de execução do programa.

Apesar de o coletor de memória baseado em regiões (gcc_crbr) apresentar um valor de alocamento baixo, quando comparado aos demais, o valor indicado na Tabela 4 mostra

o tamanho da memória alocada para região global, ou seja, a contagem de referências somente gerenciou essa parte do *heap*. O restante foi determinado durante a compilação, onde o compilador utiliza dinamicamente as regiões, sem necessidade de outro tipo de controle em tempo de execução. Além disso, o valor da memória global e a memória gerenciada por região de memória muda de acordo com o programa.

A Figura 10 apresenta o *profiling* do uso da memória na Máquina 1, de modo que esse *profiling* observa toda a memória usada na máquina e não somente para a execução do programa como apresentado na Tabela 4. A maior parte da memória disponível na máquina foi utilizada, bem como os demais coletores. Porém, a maior parte foi utilizada automaticamente sem qualquer manipulação ou varredura durante o tempo de execução. Logo, após os três segundos de execução, o programa é finalizado e a memória é liberada de volta para o sistema operacional.

Figura 10 – Uso de memória do coletor por região de memória e contagem de referências.




**Fonte:** o autor.

O mesmo comportamento se repete nas demais máquinas: a coleta por contagem de referências apresentou os melhores resultados de tempo de execução e não precisou realizar nenhuma pausa completa do programa para realizar a coleta de dados. O *benchmark* de pausas usa o envio de mensagem para armazenar os novos dados no *array* e, nesse

momento, uma nova alocação é necessária. Quando o *heap* já se encontra cheio, é necessário que ocorra uma coleta para liberação de espaço. Os valores de tempo para os coletores com contagem de referências são baixos, demonstrando que, com essa abordagem, o reúso da memória é feito de maneira dinâmica e conjunta com a execução do programa.

Como todos os coletores são concorrentes, com um mutador auxiliar, foi realizado o mesmo experimento com uso de um único núcleo (*single process*), forçando que todo o processo de coleta concorra com o programa por tempo de CPU. Os resultados são apresentados na Tabela 5.

Tabela 5 – Resultados para o *benchmark* de pausas com um único processo.

| **Coletor** | **Tempo de execução** | **Pior tempo de envio** | **Total de mem. alocada** | **Nº de pausas** |
|---|---|---|---|---|
| **Máquina 1** | | | | |
| go_ms | 1,33$s$ | 9290$us$ | 1,02$GB$ | 11 |
| gcc_ms | 3,40$s$ | 47240$us$ | 1,05$GB$ | 11 |
| go_cr std | 1,94$s$ | 826$us$ | 1,5$GB$ | 0 |
| go_cr | 1,47$s$ | 635,48$us$ | 1,02$GB$ | 0 |
| gcc_crbr | 3,37$s$ | 63,17$us$ | 24,82$KB$ | 0 |
| **Máquina 2** | | | | |
| go_ms | 612,57$ms$ | 5250$us$ | 1,02$GB$ | 11 |
| gcc_ms | 269000$ms$ | 22,92$us$ | 1,05$GB$ | 11 |
| go_cr std | 840,7$ms$ | 351,3$us$ | 1,53$GB$ | 0 |
| go_cr | 646,11$ms$ | 270,98$us$ | 1,02$GB$ | 0 |
| gcc_crbr | 295000$ms$ | 21,93$us$ | 27,20$KB$ | 0 |
| **Máquina 3** | | | | |
| go_ms | 633,89$ms$ | 4,82$ms$ | 1,02$GB$ | 11 |
| gcc_ms | 280000$ms$ | 19,98$ms$ | 1,05$GB$ | 11 |
| go_cr std | 921,7$ms$ | 6,43$ms$ | 1,5$GB$ | 0 |
| go_cr | 709,64$ms$ | 4,95$ms$ | 1,02$GB$ | 0 |
| gcc_crbr | 311000$ms$ | 6,7$ms$ | 27,19$KB$ | 0 |

É visto que o mesmo experimento, com um único processo disponível, não degrada todos os coletores; alguns coletores como o MS chega até a melhorar o tempo total da execução. O *benchmark* de pausas por si só não explora a concorrência total das arquiteturas, dada a baixa mudança nos resultados. Além disso, observa-se que o único processo concorrente é a coleta de dados. O coletor de CR sofreu uma pequena degradação, mas o número de pausas e o montante de memória utilizada foi o mesmo. O tempo de envio nesse *benchmark* representa o tempo de coleta.

### 5.4.2 Binary-tree

O *benchmark binary-tree* (FULGHAM; GOUY, 2009) gera uma grande carga de trabalho para o coletor de lixo, alocando árvores diretamente na memória e percorrendo todos os nós para verificar se a mesma existe e, desalocando repetidas vezes, esse *benchmark* recebe como entrada o número de níveis que a árvore deve ter. Como padrão, são adotados os valores de entrada sendo: 7, 14 e 21.

#### 5.4.2.1 Máquina 1

Para os testes apresentados daqui em diante, foram realizadas 300 execuções dos experimentos para cada entrada nas três diferentes arquiteturas usadas. Os resultados são ilustrados nos gráficos a seguir. Os valores apresentados dizem respeito ao tempo de execução em milissegundo.

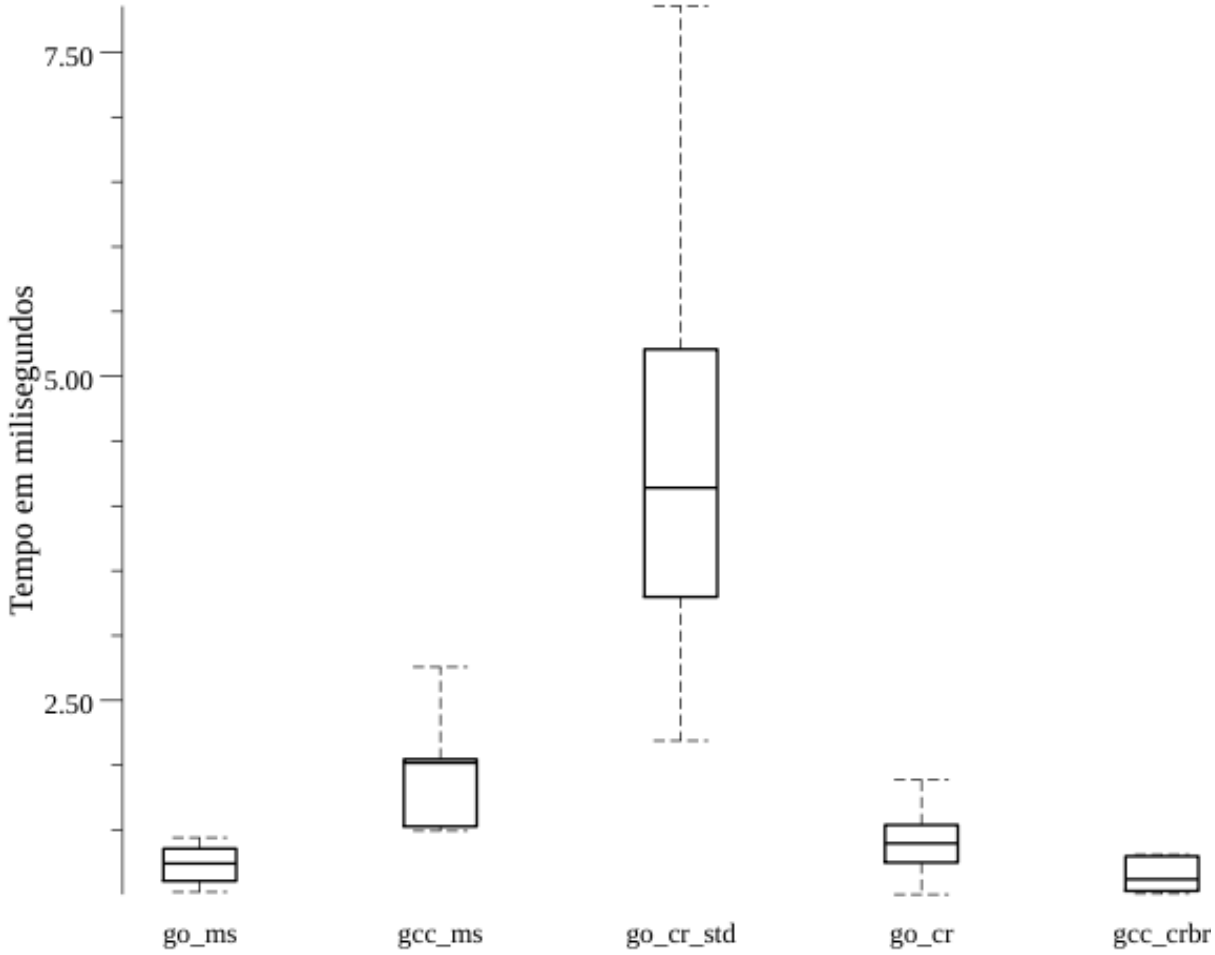
Na Figura 11, a CR baseada e regiões de memória apresentou menor tempo de execução que todos os demais coletores, devido a maior parte da coleta ser predeterminada em tempo de compilação as pausas são reduzidas ao mínimo ou até mesmo não ocorrem, gerando benefícios a execução do programa. Já a CR padrão apresentada em Formiga (2011) levou mais tempo para a execução completa do programa, como a CR impõe atualizações de todos os ponteiros ligados ao objeto a ser liberado, essas operações podem adicionar uma maior carga de trabalho durante as coletas.

Com valor de entrada igual a 14 podemos observar que os valores de todos os coletores apresentam menor variação, por essa razão os valores dos quartis na Figura 12 não se destacam, mas podemos ver que com a maior carga de trabalho, consequentemente maior uso da memória, o tempo de execução é impactado com a ação dos coletores.

A coleta por CR supera o coletor nativo de Go, reduzindo o tempo de execução, mas a mesma coleta com base em regiões de memória continua com menor tempo de execução que os demais, e em último a CR padrão.

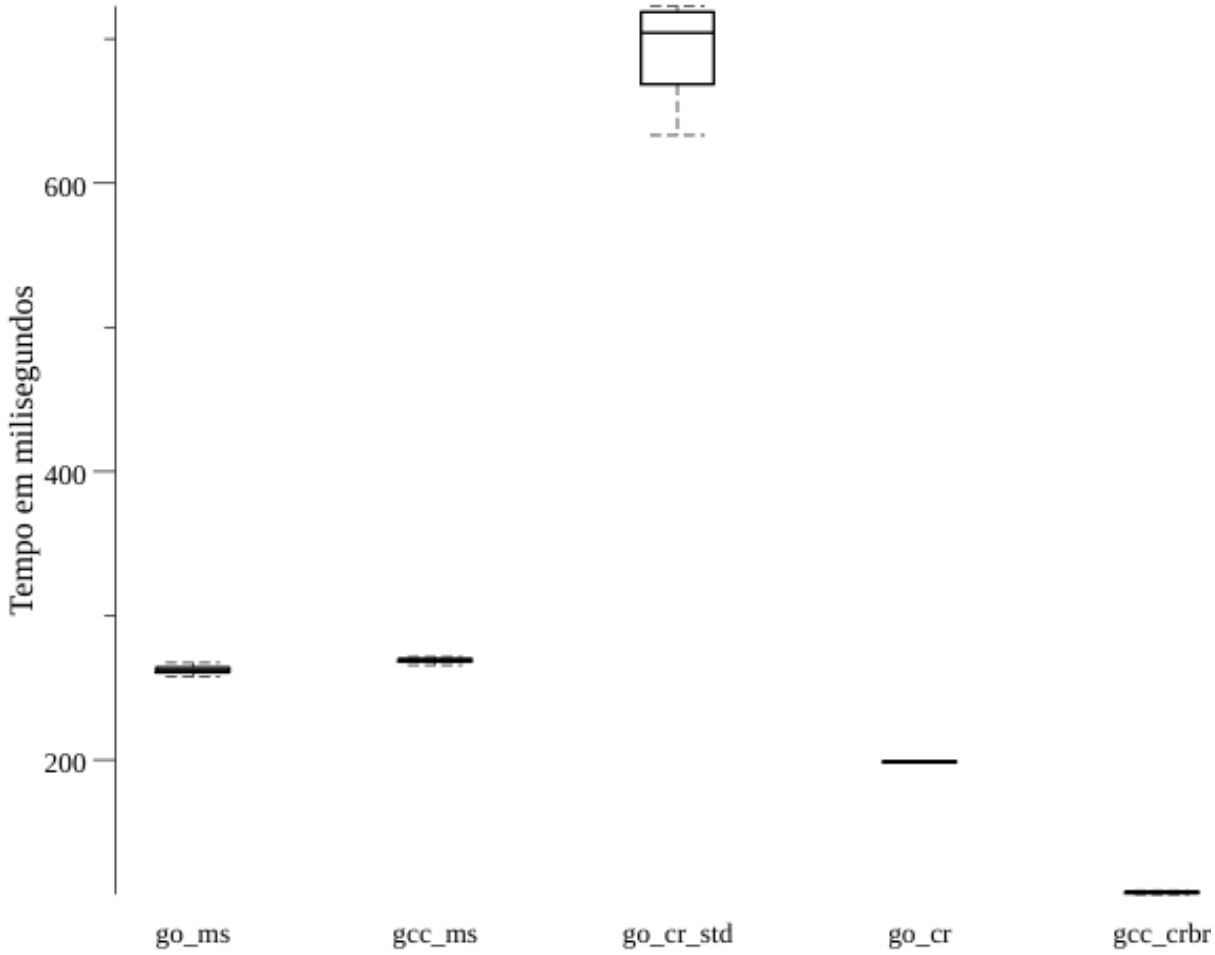
O coletor por contagem de referência (`go_cr`) apresentam similar desempenho ao coletor nativo proposto pela linguagem Go, no que diz respeito ao tempo de execução total do *benchmark*. Por esta semelhança, principalmente entre os coletores `go_ms` e `go_cr` apresentamos os valores na tabela abaixo.

Figura 11 – Binary-tree Máquina 1 com entrada igual a 7.




**Fonte:** o autor.

Figura 12 – Binary-tree Máquina 1 com entrada igual a 14.




**Fonte:** o autor.

Tabela 6 – Resultados para *benchmark* Binary-tree na Máquina 1.

| **Entrada 7** | | | |
|---|---|---|---|
| **Coletor** | go_ms | | go_cr |
| **Tempo de execução** | 1,23$ms$ | | 1,40$ms$ |
| **Mem alocada** | 0,20$MB$ | | 0,20$MB$ |
| **Mem liberada** | $\approx 0\%$ | | $\approx 95\%$ |
| **Número de pausas** | 0 | | 0 |
| **Desvio padrão tm** | 0,15 | | 0,24 |
| **Desvio padrão Mem** | 57,53 | | 38,56 |
| | 0,05 | 0,00 | |
| ***p-value* Tempo/Mem** | 0,1 | 0,00 | |
| | 0,95 | 0,99 | |
| **Entrada 14** | | | |
| **Tempo de execução** | 261,98$ms$ | | 198,69$ms$ |
| **Mem alocada** | 51,62$MB$ | | 51,61$MB$ |
| **Mem liberada** | $\approx 75\%$ | | $\approx 98\%$ |
| **Número de pausas** | 14 | | 0 |
| **Desvio padrão tm** | 2,77 | | 0,4 |
| **Desvio padrão Mem** | 273,26 | | 22,3 |
| | 1 | 1 | |
| ***p-value* Tempo/Mem** | 0,00 | 0,00 | |
| | 0,00 | 0,00 | |

A Tabela 6 mostra a média do tempo de execução e a média da memória alocada no *heap*. Nota-se o motivo de os dados serem semelhantes no gráfico, ainda mais dada a grande variedade da escala usada no gráfico para compor o resultados das três entradas usadas.

O *benchmark* com entrada igual a 7 não apresentou nenhum ciclo de coleta, logo, não foi liberado nenhum objeto do *heap*. No entanto, como a coleta por contagem de referência os dados são liberados à medida que o programa é executado, ou seja, sempre há coleta, por esse motivo, o valor de coleta é superior.

Os resultados estatísticos mostram que para hipótese em que os valores das médias dos tempo de execução do coletor MS são menores que a coleta por CR é confirmada, em termos dos valores de cada *fold*, isso indica que o coleta por CR foi menos eficiente para o tempo de execução com a entrada igual a 7.

Quato aos valores da segunda entrada do *benchmark* (entrada igual a 14), o principal resultado é que podemos ver que, à medida que o número de ciclos de coleta aumenta na coleta por MS, o valor de tempo de execução sofre uma degradação. O número de ciclos e pausas do coletor, as hipóteses também alteram, mostrando que, para a hipótese da primeira entrada ser menor que a segunda é negada, tanto para o tempo de execução como para uso de memória.

Para essa arquitetura, não foi possível concluir as execuções para entrada de 21 nós para o algoritmo com o coletor de CR, o sistema ficou sem memória para completar sua execução.

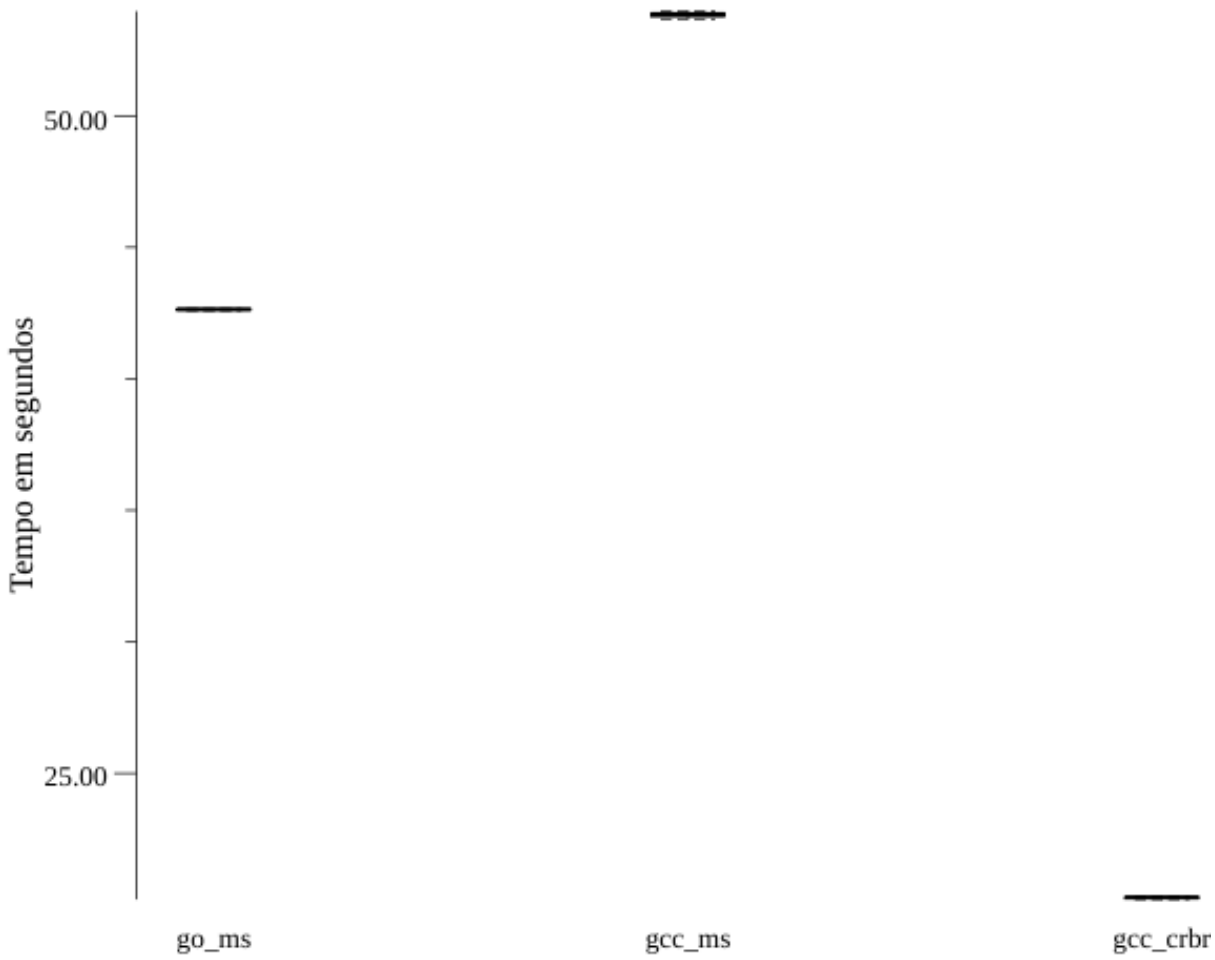
Os valores foram obtidos através da média dos resultados dos experimentos com cada entrada proposta para esse *benchmark*, porém, não foi possível realizar o experimento com entrada de 21 com o coletor de CR para essa arquitetura, isso se deu por falta de memória suficiente. Ambos os coletores (CR e MS) fizeram maior uso de memória, enquanto os coletores no compilador nativo para GCC foram capazes de otimizar o valor alocado. O algoritmo MS obteve maior tempo de execução com o compilador para GCC.

A discrepância do coletor com contagem de referências baseada em regiões de memória é claramente observada, tanto em relação ao tempo de execução quanto ao uso de memória alocada. Como visto anteriormente, os valores expressos nas Figuras 11, 12 e 13 representam o valor da região global de alocamento, que foi constante em 27.49 $KB$ para todas as entradas nesse experimento.

Os resultados sobre a coleta por CR aqui apresentados reportam aos valores obtidos para a abordagem descrita nessa tese, quando comparado ao experimento *Binary-tree* entre a versão original da CR com detecção de ciclos criada por Formiga (2011) e a descrita nessa tese, os valores de tempo de execução e uso de memória na Máquina 1 são:

- Entrada igual a 7: 4,65$ms$, 0,34$MB$ coleta de $\approx$ 99%
- Entrada igual a 14: 702,52$ms$, 103,16$MB$ coleta de $\approx$ 99%

Figura 13 – Binary-tree Máquina 1 com entrada igual a 21.




**Fonte:** o autor.

A proposta apresentada nesta tese da CR toma 30% do tempo e consome cerca de 57% da memória comparada ao algoritmo original, isso para entrada igual a 7. Essa média também é observada com a entrada igual a 14. Ambos não foram capazes de finalizar o teste com entrada igual a 21.

#### 5.4.2.2 Máquina 2

No segundo cenário apresentado na Tabela 7 e nas Figuras 14, 15 e 16 para o mesmo experimento, porém para as entradas 7, 14 e 21 respectivamente. As figuras buscam mostrar a distribuição empírica do dados em relação ao tempo de execução.

Pode-se observar que a contagem de referências deu um salto no desempenho quando comparado aos coletores proposto em Golang, isso é visivelmente notado principalmente nas Figuras 14 e 15.

A variação dos resultados obtidos diminui, isso é observado nas Figuras 15 e 16, que

Tabela 7 – Resultados para *benchmark* Binary-tree para Máquina 2.

| **Coletor** | **Tempo de execução** | **Total de mem. alocada** | **Mem liberada** | **Nº de pausas** |
|---|---|---|---|---|
| | | **Entrada 7** | | |
| go_ms | 468,6$us$ | 0,21$MB$ | $\approx 0\%$ | 0 |
| gcc_ms | 604,1$us$ | 0,17$MB$ | $\approx 16,8\%$ | 5 |
| go_cr std | 104000$us$ | 0,34$MB$ | $\approx 99,0\%$ | 0 |
| go_cr | 375,71$us$ | 0,20$MB$ | $\approx 95,0\%$ | 0 |
| gcc_crbr | 187,07$us$ | 22,2$KB$ | $\approx 99,0\%$ | 0 |
| | | **Entrada 14** | | |
| go_ms | 147,85$ms$ | 51,63$MB$ | $\approx 5,0\%$ | 14 |
| gcc_ms | 201,94$ms$ | 51,80$MB$ | $\approx 2,0\%$ | 72 |
| go_cr std | 347,41$ms$ | 103,17$MB$ | $\approx 99\%$ | 0 |
| go_cr | 125,13$ms$ | 51,62$MB$ | $\approx 97,7\%$ | 0 |
| gcc_crbr | 60,86$ms$ | 24,2$KB$ | $\approx 99\%$ | 0 |
| | | **Entrada 21** | | |
| go_ms | 29,81$s$ | 9,82$GB$ | $\approx 1,36\%$ | 145 |
| gcc_ms | 39,37$s$ | 9,84$GB$ | $\approx 1,37\%$ | 125 |
| go_cr std | - | - | - | - |
| go_cr | 65$s$ | 9,82$GB$ | $\approx 99\%$ | 4 |
| gcc_crbr | 12,13$s$ | 24,60$KB$ | $\approx 99\%$ | 1 |

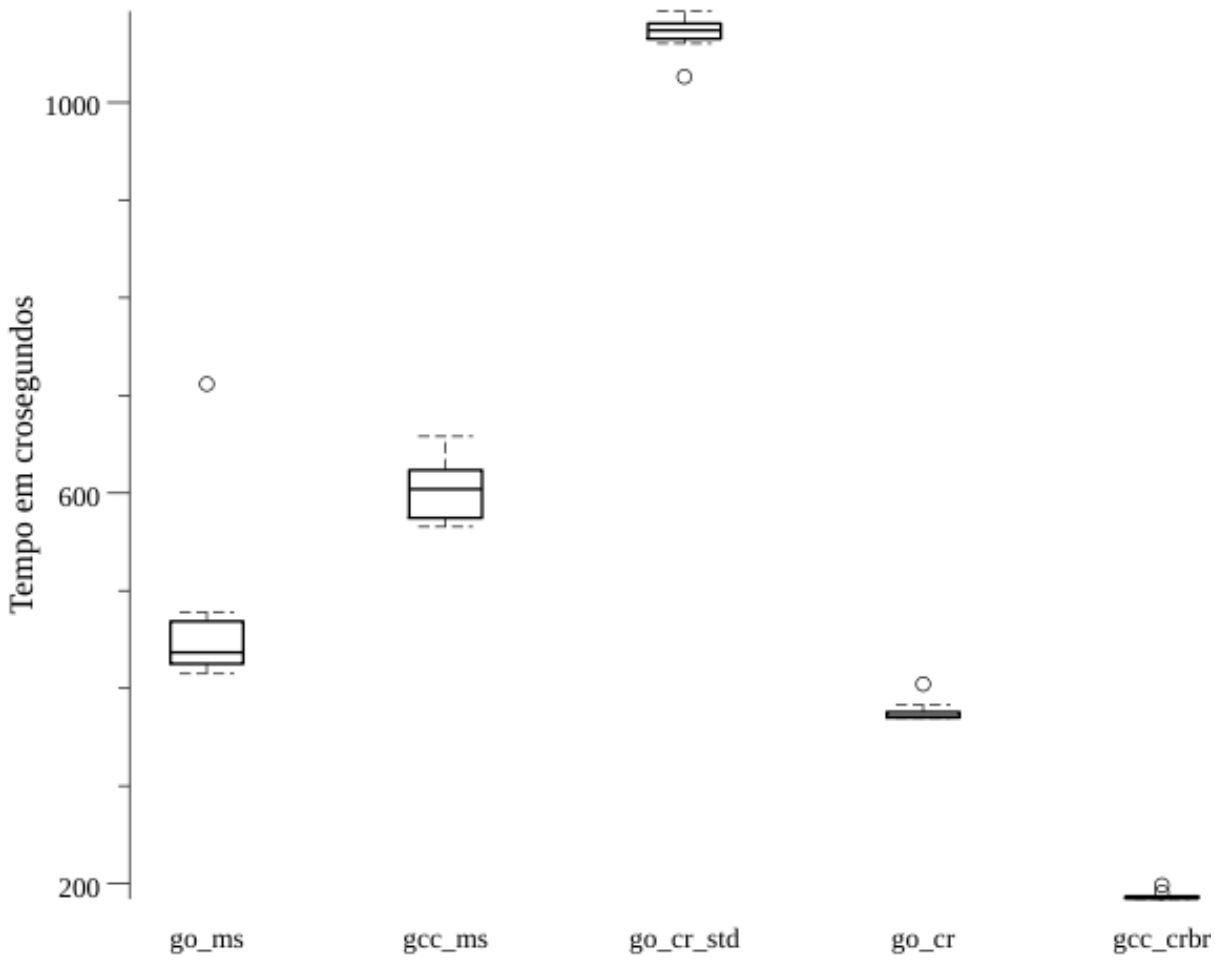
não chegam a apresentar diferença entre os valores dos quartis, isso se dá a medida que se aumenta a carga de trabalho, além disso os resultados apresentam baixo valor de *outliers*, também com o aumento da carga na execução.

Embora os coletores com MS tenham realizado diversas pausas, impactando diretamente no tempo de execução do programa, ainda assim, o coletor com base em regiões de memória apresentou um valor distante em relação a todos os demais, isso para todos os casos desse *benchmark* na Máquina 2.

Os resultados apresentados nas figuras dizem respeito unicamente ao tempo de execução total do *benchmark*, os valores em relação ao consumo de memória e pausas realizadas para coleta são descritas na tabela a seguir.

Enquanto a contagem de referências tende a cair, o desempenho degradando os valores à medida que o uso de memória sobe é dado pela necessidade de coleta por estruturas cíclicas presente, essa relação entre o tempo de execução, uso de memória e número de pausa é melhor apresentado na Tabela 7. Isso pode indicar que a contagem de referências apresentou menor desempenho na última sessão dos teste, ou seja, com valor de entrada igual a 21, obtendo um maior custo de tempo de execução. Porém vale se ressaltar que a CR apresenta um alto índice de taxa de coleta, que quer dizer maior trabalho e presença

Figura 14 – Binary-tree Máquina 2 com entrada igual a 7.



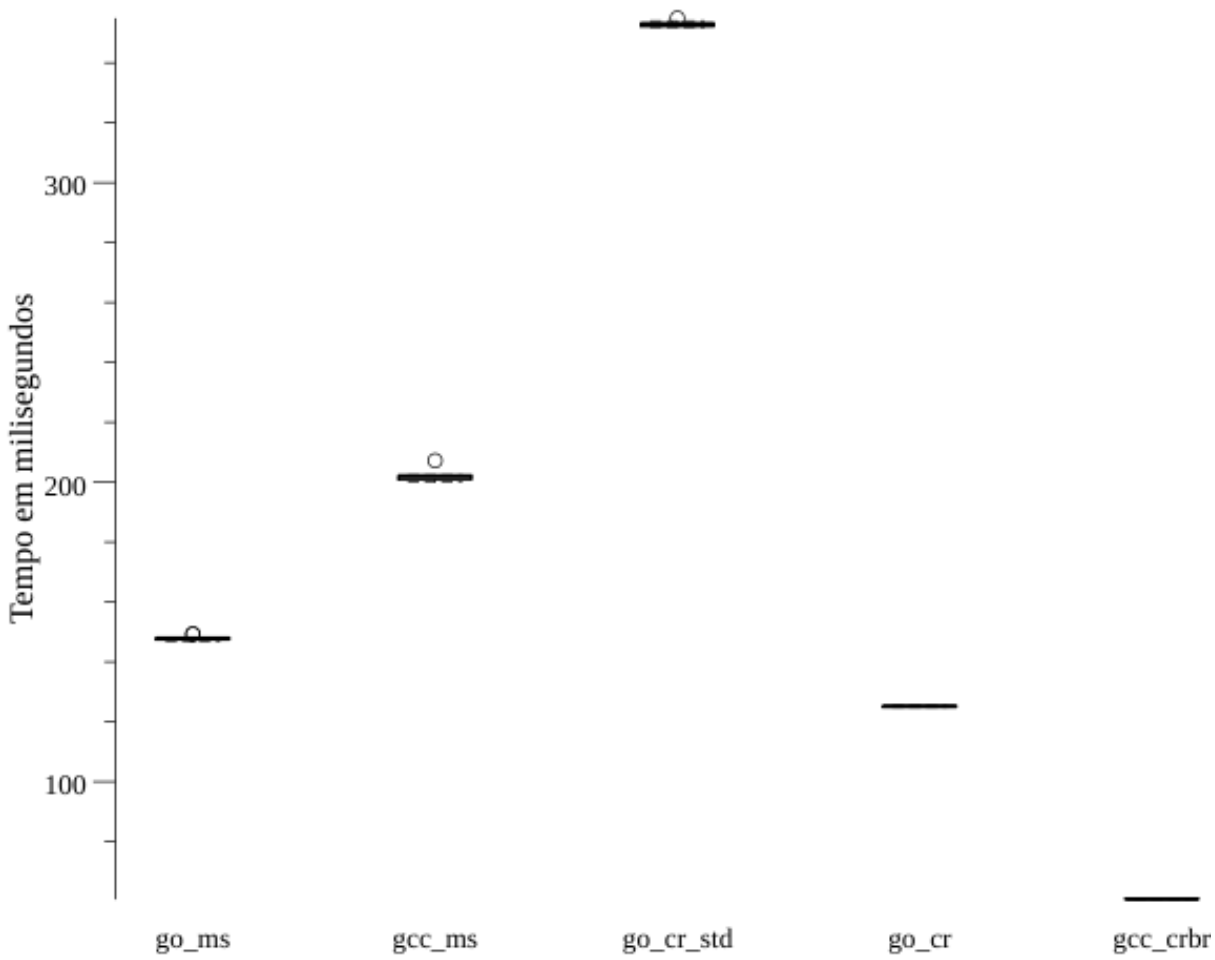

**Fonte:** o autor.

da atuação do coletor, isso reforça o que a literatura diz: a sempre um momento ótimo para a ação do coletor, a degradação apresentada com a CR com a entrada 21 pode significar que esse momento ótimo para essa arquitetura já passou.

Outro dado que pode ser observado na Tabela 7 sobre execução desse experimento, são: memória e pausa, além disso mostra que foi possível reduzir o valor de uso de memória, bem como o valor de número de pausas elevados porém, não relatado no gráfico. À medida que ocorre mais uso de memória, os coletores que usam MS realizam diversas pausas para coleta variando os valores de acordo com a entrada. A ordem de grandeza no tempo de execução a medida que se aumenta o dado de entrada mostra o aumento da carga de trabalho, o *benchmark* é o mesmo porém a criação de novos nós na árvore impactam diretamente na execução do experimento.

Para o primeiro caso, não são observadas pausas para o compilador em Go, e poucas paradas são realizadas pelo compilador GCC, nisso pode-se observar que mesmo usando

Figura 15 – Binary-tree Máquina com entrada igual a 14.



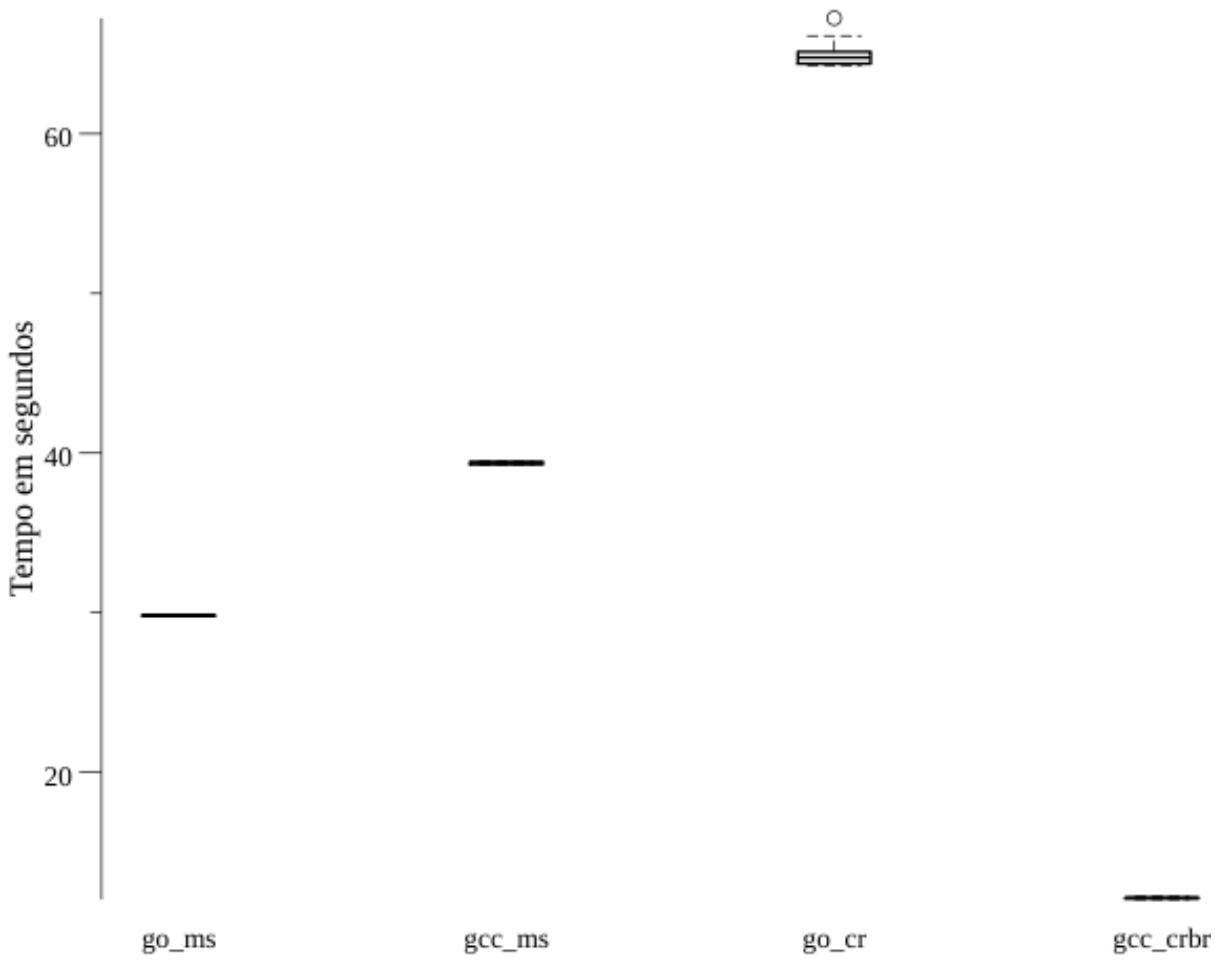

**Fonte:** o autor.

a mesma abordagem de coleta os compiladores de Go apresentam divergência na manipulação do uso da memória, isso gera um custo na performance do tempo e memória para todas as entradas em praticamente todas as arquiteturas contempladas nesse experimento.

Com a variação da entrada para 14 nós são realizadas também 14 paradas pelos coletores de Go, enquanto o coletor GCC apresenta 72 ciclos, resultando em 72 paradas completas do programa média cresce para a entrada igual a 21, ambos os coletores com MS. Os tempos de pausas foram de $\approx$ 0,79 $ms$ para entrada com valor 14 e $\approx$ 9,2 $ms$ para a entrada igual a 21.

Por sua vez, a coleta por CR com o compilador nativo de Go somente realiza 4 pausas com entrada de 21 nós, enquanto o coletor por CR com base em regiões de memória só realiza uma única parada em média para coleta de estruturas em ciclo, isso para a maior carga de uso de trabalho do teste. Por outro lado a CR original Formiga (2011) não foi capaz de finalizar a execução do *benchmark* com entrada igual a 21 nessa arquitetura,

Figura 16 – Binary-tree Máquina com entrada igual a 21.




**Fonte:** o autor.

isso devido a grande quantidade de memória utilizada, o mesmo não foi capaz de liberar memória suficiente para que a execução fosse finalizada.

#### 5.4.2.3 Máquina 3

Os resultados do *benchmark Binary-tree* na última arquitetura são apresentados em seguida. Os valores apresentados em média apresentam uma variação dos resultados, mas é possível verificar o impacto dos valores resultantes devido à maior quantidade de memória disponível na máquina *host*.

Quando são equiparados, os coletores por CR em Go, pode-se verificar a redução do uso total de memória acumulada e maior performance em tempo de execução. O coletor de CR implementado sem as melhorias não foi capaz de realizar a execução com entrada igual a 21, por não ter memória suficiente.

A Tabela 8 mostra um resultado diferente até então visto. Isso demonstra a influência

Tabela 8 – Resultados para *benchmark* Binary-tree para Máquina 3.

| **Coletor** | **Tempo de execução** | **Total de mem. alocada** | **Mem liberada** | **Nº de pausas** |
|---|---|---|---|---|
| **Entrada 7** | | | | |
| go_ms | 345,34$us$ | 204,92$KB$ | $\approx 0\%$ | 0 |
| gcc_ms | 195000$us$ | 174,23$KB$ | $\approx 16,7\%$ | 4 |
| go_cr std | 465000$us$ | 307,77$KB$ | $\approx 99\%$ | 0 |
| go_cr | 346,94$us$ | 205,27$KB$ | $\approx 97,0\%$ | 0 |
| gcc_crbr | 197,7$us$ | 22,25$KB$ | $\approx 87,7\%$ | 0 |
| **Entrada 14** | | | | |
| go_ms | 129,13$ms$ | 51,63$MB$ | $\approx 5,08\%$ | 14 |
| gcc_ms | 269,39$ms$ | 47,13$MB$ | $\approx 2,03\%$ | 72 |
| go_cr std | 702,53$ms$ | 103,16$MB$ | $\approx 99\%$ | 0 |
| go_cr | 124,28$ms$ | 51,62$MB$ | $\approx 98\%$ | 0 |
| gcc_crbr | 58,57$ms$ | 22,21$KB$ | $\approx 96,4\%$ | 0 |
| **Entrada 21** | | | | |
| go_ms | 30,15$ms$ | 9,79$GB$ | $\approx 1,02\%$ | 144 |
| gcc_ms | 53,83$ms$ | 9,84$GB$ | $\approx 2,05\%$ | 124 |
| go_cr std | - | - | - | - |
| go_cr | 47,39$ms$ | 10,6$GB$ | $\approx 98,0\%$ | 4 |
| gcc_crbr | 20,27$ms$ | 24,6$KB$ | $\approx 87,0\%$ | 1 |

da arquitetura e recursos disponíveis que alteram o desempenho dos programas. Devido à maior quantidade de memória disponível, os coletores por MS apresentam elevação no desempenho. Porém o compilador GCC de Go apresentou uma degradação no tempo de execução nessa arquitetura, quando comparado aos resultados anteriores.

A coleta com o compilador de Go com uso da CR é bastante similar até então, mas, para todos os casos, o uso da CR com base em regiões de memória apresentou o melhor desempenho em tempo de processamento. O uso de memória no *heap* também apresentou valores baixos, mas, como visto anteriormente esse valor é referente ao valor gerenciado em tempo de execução.

Maiores informações são apresentadas na Tabela 8, o compilador de Go com CR reduz o desempenho antes apresentado devido à alta taxa de coleta e consumo de memória, mas nota-se que isso ocorre principalmente no último segmento do teste, onde foram desempenhadas 4 pausas para varredura de estruturas em ciclo.

O compilador com base em regiões de memória apresentou os melhores desempenhos ao longo de todos os testes até aqui apresentados. É possível ver que a região global apresentou uma pausa por ciclos na região global para maior carga de trabalho e consumo de memória.

Para todos os resultados até então apresentados, nota-se a diferença produzida pelos compiladores Go e GCC. Apesar da diferença de resultados, o tamanho do código binário gerado por todos os compiladores aqui utilizados não apresentou grande diferença em relação ao tamanho dos arquivos gerados.

Nota-se que o uso da coleta por CR implica em maior coleta de dados, já que a liberação de objetos na *heap* ocorre concomitantemente com a execução do programa, o que é perceptível em ambos os coletores por CR em Go. Mais uma vez o coletor baseado no algoritmo original de Formiga (2011) gasta cerca de 70% a mais do tempo de processamento e mais de 50% do uso de memória em relação a proposta apresentada nesta tese.

### 5.4.3 Matmul

Com pouca alocação e vários *loops* aninhados para multiplicar duas matrizes, o *benchmark* *matmul*, também utilizado em (DAVIS, 2015), busca avaliar o desempenho de correspondência de expressões regulares no contexto de análises de sequências biológicas. O mesmo apresenta diferentes entradas, aqui propostas, variando de 100, 1.000 até 5.000.

#### 5.4.3.1 Máquina 1

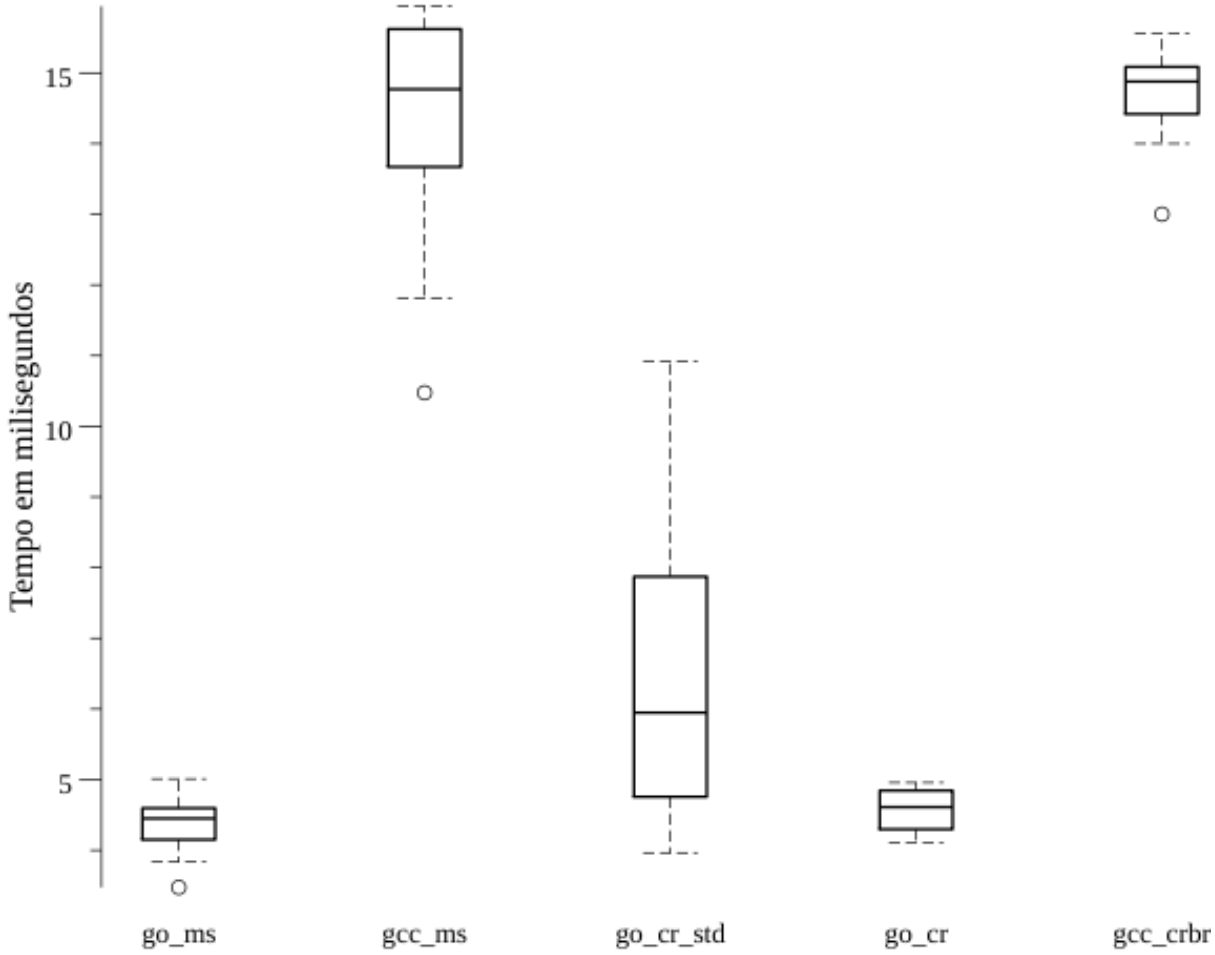
Mais uma vez a coleta por contagem de referências mostra-se semelhante em desempenho, e isso pode ser observado em todas as Figuras 19, 20 e 21, onde é apresentado os valores no aspecto de tempo de execução. O coletor de MS com o compilador *front-end* GCC foi o que tomou mais tempo para finalização das tarefas. Aqui, nota-se que o compilador de GCC com contagem de referências baseado em regiões de memória não mais apresentou ganhos em tempo de execução como nos teste anteriores. Isso é caracterizado porque esse *benchmark* realiza poucas alocações e mantém os objetos no *heap* por mais tempo segurando, o que acaba por liberar pouca região de memória para contagem de referências e pouco reúso de sub-regiões de memória, impactando o desempenho da execução do programa. Comportamento similar já havia sido reportado em Davis (2015).

Ambas as coletas por CR desempenharam melhor performance que os coletores com o compilador Gccgo, onde a CR com o algoritmo original de Formiga (2011) é representada como `go_cr_std`.

Por apresentar similaridade nos resultados em ambos os coletores do compilador em Go, são apresentados na Tabela 9 as médias obtidas e os resultados estatísticos para elucidação dos resultados.

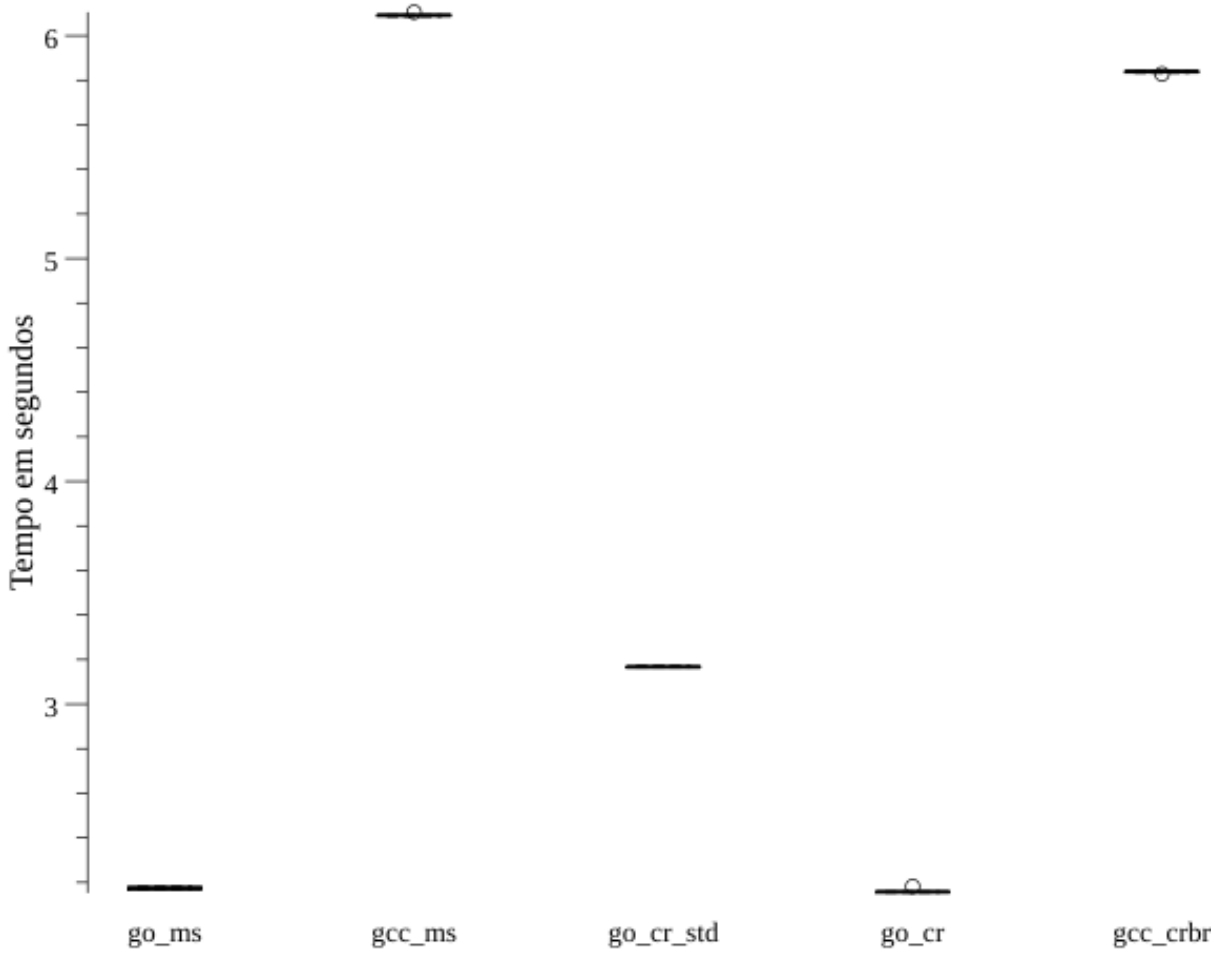
Baseado nos valor das médias de tempo, a coleta por MS foi mais eficiente, porém, a coleta por CR usou menos memória. Os dados estatísticos mostram que, de fato, há uma diferença entre os tempos de execuções apresentados e que a amostra da coleta por MS é menor que a coleta por CR, em tempo de execução e consumo de memória, porém com base no valor da memória liberada, houve pouca ou até mesmo nenhuma coleta com o compilador de MS, enquanto a CR liberou mais espaço no *heap*.

Figura 17 – Matmul com entrada igual a 100 na Máquina 1.



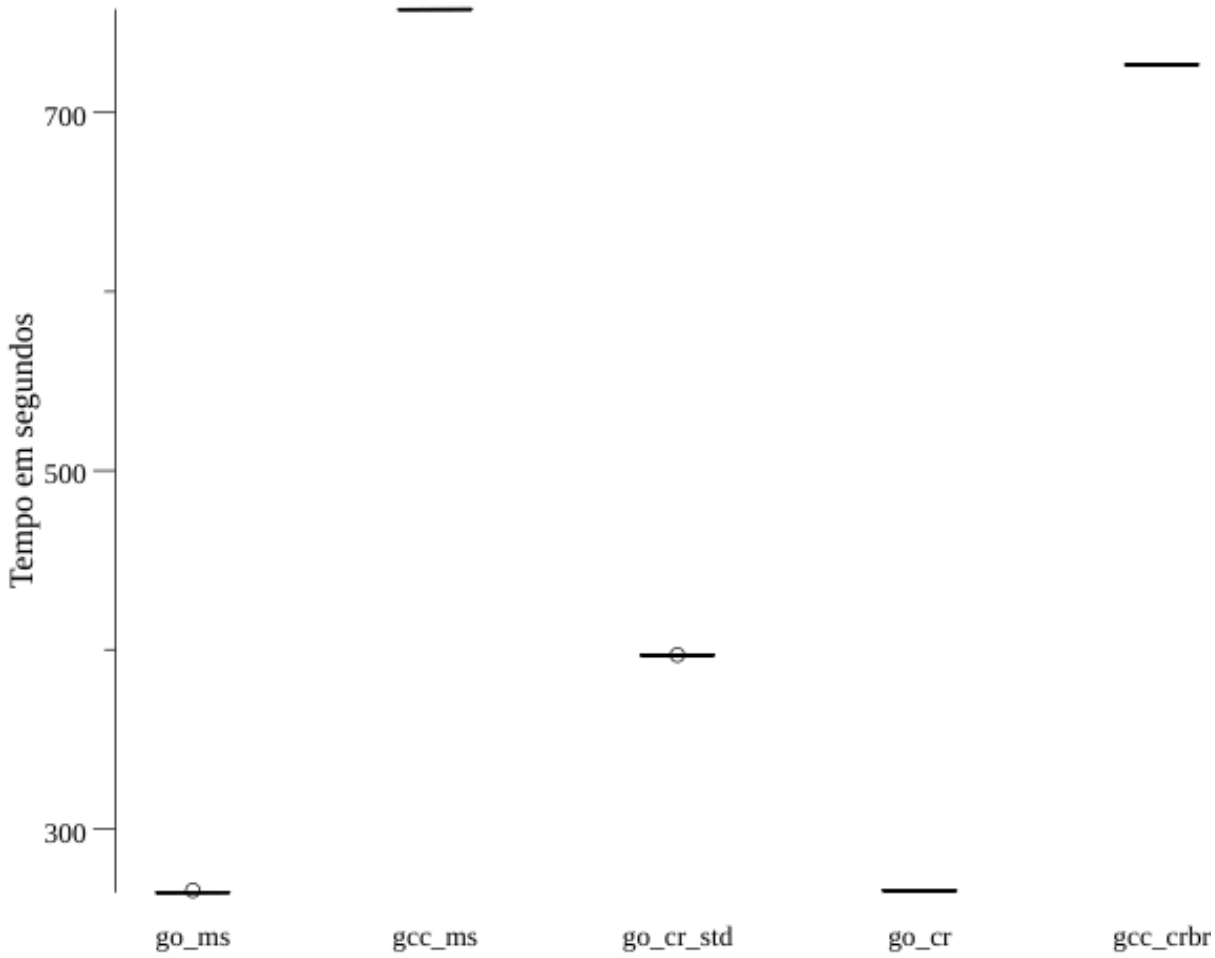

**Fonte:** o autor.

Figura 18 – Matmul com entrada igual a 1000 na Máquina 1.




**Fonte:** o autor.

Figura 19 – Matmul com entrada igual a 5000 na Máquina 1.




**Fonte:** o autor.

Tabela 9 – Resultados para *benchmark* Matmul na Máquina 1.

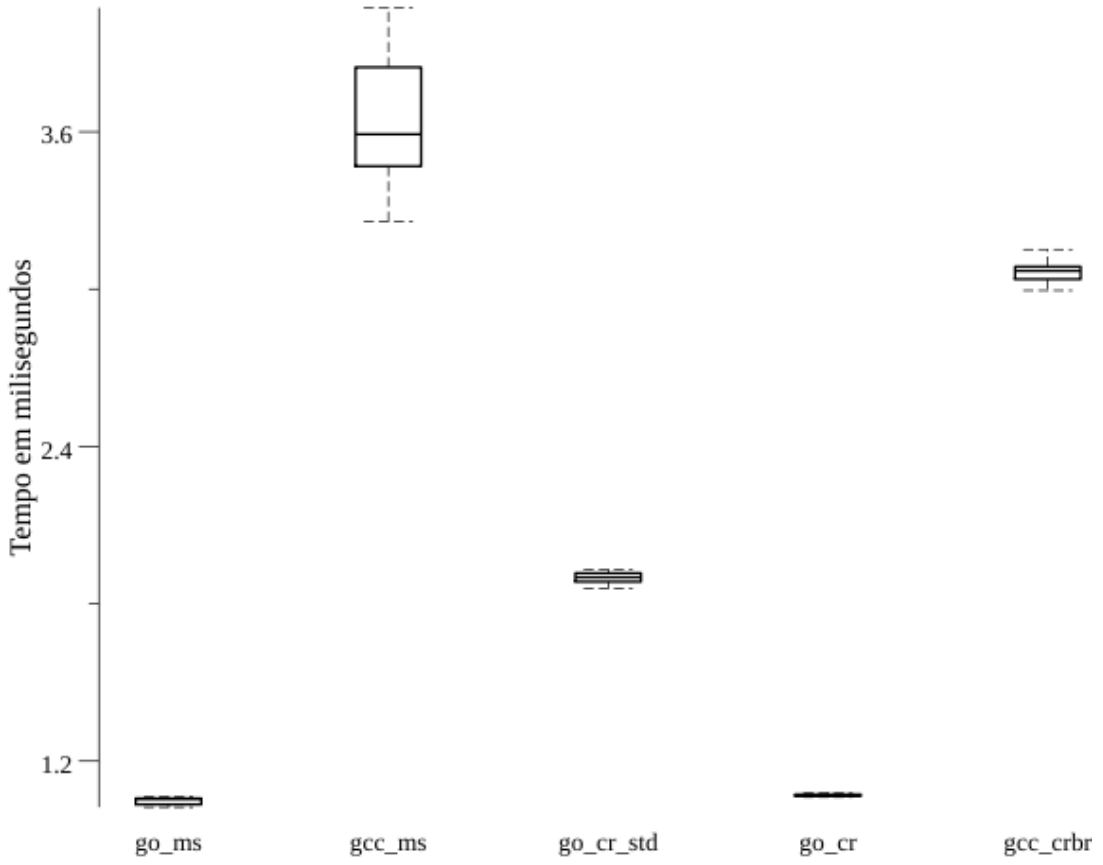
| **Entrada 100** | | |
|---|---|---|
| **Coletor** | go_ms | go_cr |
| **Tempo de execução** | 4,38$ms$ | 4,58$ms$ |
| **Mem alocada** | 47,12$MB$ | 42,41$MB$ |
| **Mem liberada** | $\approx 0\%$ | $\approx 90\%$ |
| **Número de pausas** | 0 | 0 |
| **Desvio padrão tm** | 0,47 | 0,31 |
| **Desvio padrão Mem** | 57,53 | 38,56 |
| | 0,24 0,03 | |
| ***p-value* Tempo/Mem** | 0,48 0,06 | |
| | 0,78 0,97 | |
| **Entrada 1.000** | | |
| **Tempo de execução** | 2,17$s$ | 2,16$s$ |
| **Mem alocada** | 32,93$MB$ | 32,92$MB$ |
| **Mem liberada** | $\approx 75\%$ | $\approx 98\%$ |
| **Número de pausas** | 4 | 0 |
| **Desvio padrão tm** | 0,006 | 0,008 |
| **Desvio padrão Mem** | 135,26 | 28,96 |
| | 0,99 1 | |
| ***p-value* Tempo/Mem** | 0,00 0,00 | |
| | 0,00 0,00 | |
| **Entrada 5.000** | | |
| **Tempo de execução** | 264,53$s$ | 265,45$s$ |
| **Mem alocada** | 819,75$MB$ | 819,75$MB$ |
| **Mem liberada** | $\approx 75,1\%$ | $\approx 98\%$ |
| **Número de pausas** | 10 | 0 |
| **Desvio padrão tm** | 0,36 | 0,10 |
| **Desvio padrão Mem** | 194,72 | 12,15 |
| | 0,00 1 | |
| ***p-value* Tempo/Mem** | 0,00 0,00 | |
| | 0,99 0,00 | |

Os resultados se invertem à medida que o valor de entrada aumenta. É possível ver que ocorre maior consumo de tempo e memória de ambos os coletores, porém, a partir do aumento do número de coletas, também cresce o número de pausas no uso do coletor por MS. Isso gera um impacto na performance do programa, ainda similares porém os dados estatísticos mostram que, para ambas as entradas 1000 e 5000, o tempo de execução do coletor e consumo de memória por CR é mais eficiente.

#### 5.4.3.2 Máquina 2

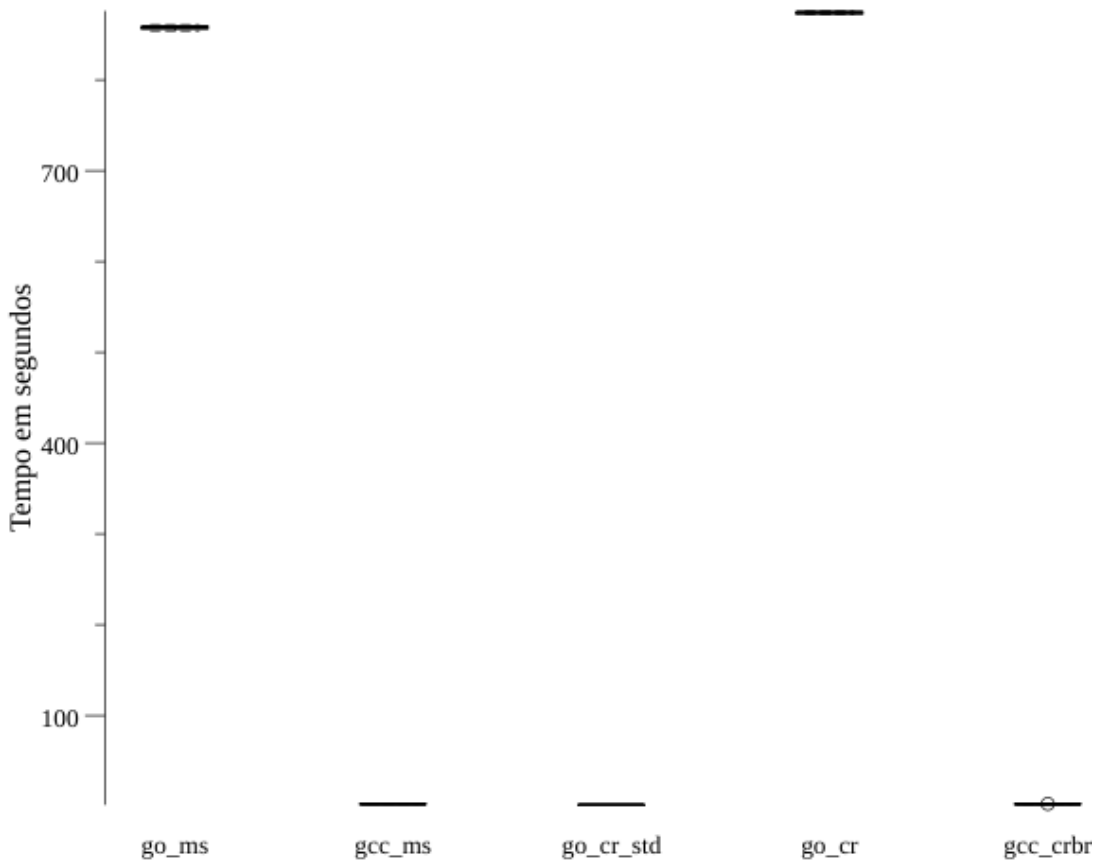
A Figura 20 ilustra os resultados do experimento na arquitetura com entrada igual a 100 com quantidade de memória igual a $8GB$. Novamente há uma proximidade dos valores resultantes dos coletores em Go, exceto pela coleta por CR original, esse semelhança também nas Figuras 21 e 22, o compilador Gogc foi o que levou maior tempo para finalização de todo o experimento, seguido do coletor com CR baseado em regiões de memória, que continua com um baixo uso de memória dinâmica, porém, dada a natureza do programa, não são observadas maiores melhorias em tempo de execução.

Figura 20 – Matmul Máquina 2 com entrada igual a 100.




**Fonte:** o autor.

Figura 21 – Matmul Máquina 2 com entrada igual a 1000.

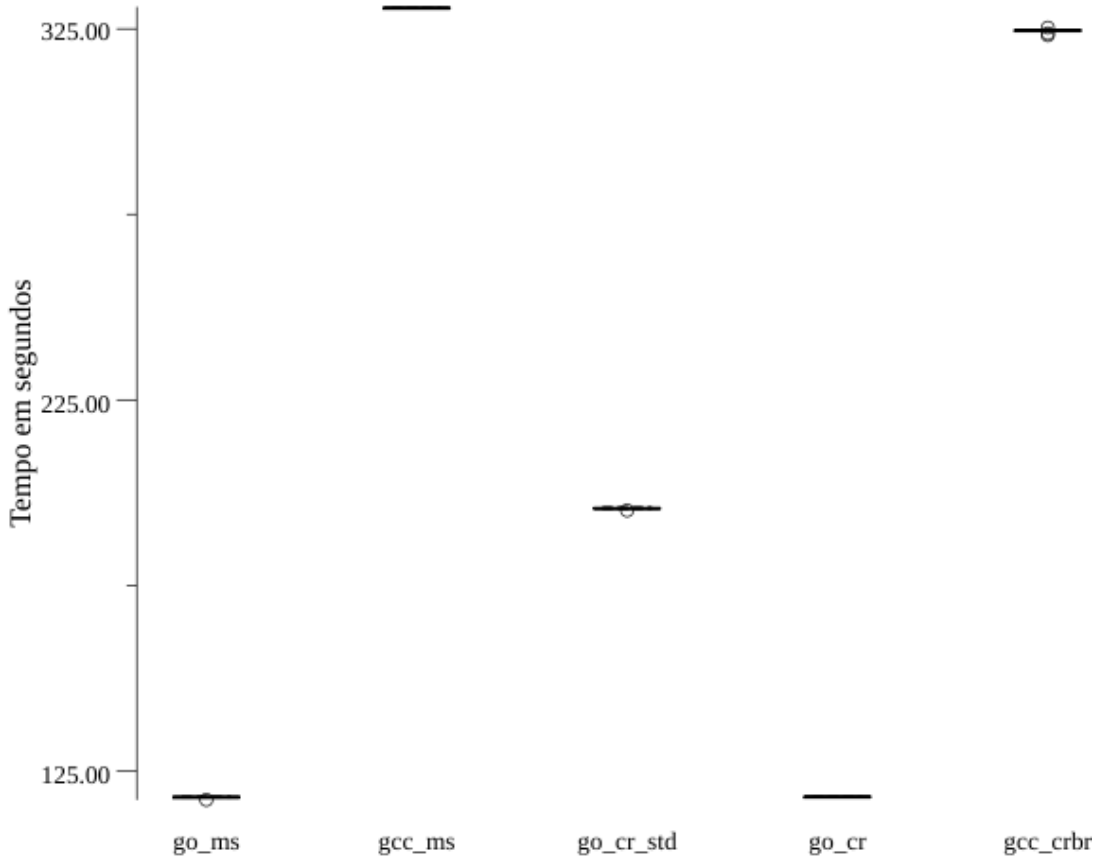



**Fonte:** o autor.

Maiores informações podem ser observadas a partir da Tabela 7, de resultados estatísticos, para a entrada inicial de 100, os valores das médias são similares, mas sem nenhuma coleta por parte do compilador com uso de MS, os valores estatísticos mostram que, para a hipótese nula ambas as amostras, tanto de tempo como de consumo de memória, divergem e, para ambos os casos, a coleta com uso de MS é mais eficiente.

Dado a segunda entrada do experimento, o número de coletas por parte da abordagem com MS aumenta, porém, ainda é baixa e o tempo de execução permanece menor em relação ao CR, mesmo com o consumo de memória sendo maior, o mesmo se observa para a última entrada de 5000 do experimento.

Figura 22 – Matmul Máquina 2 com entrada igual a 5000.




**Fonte:** o autor.

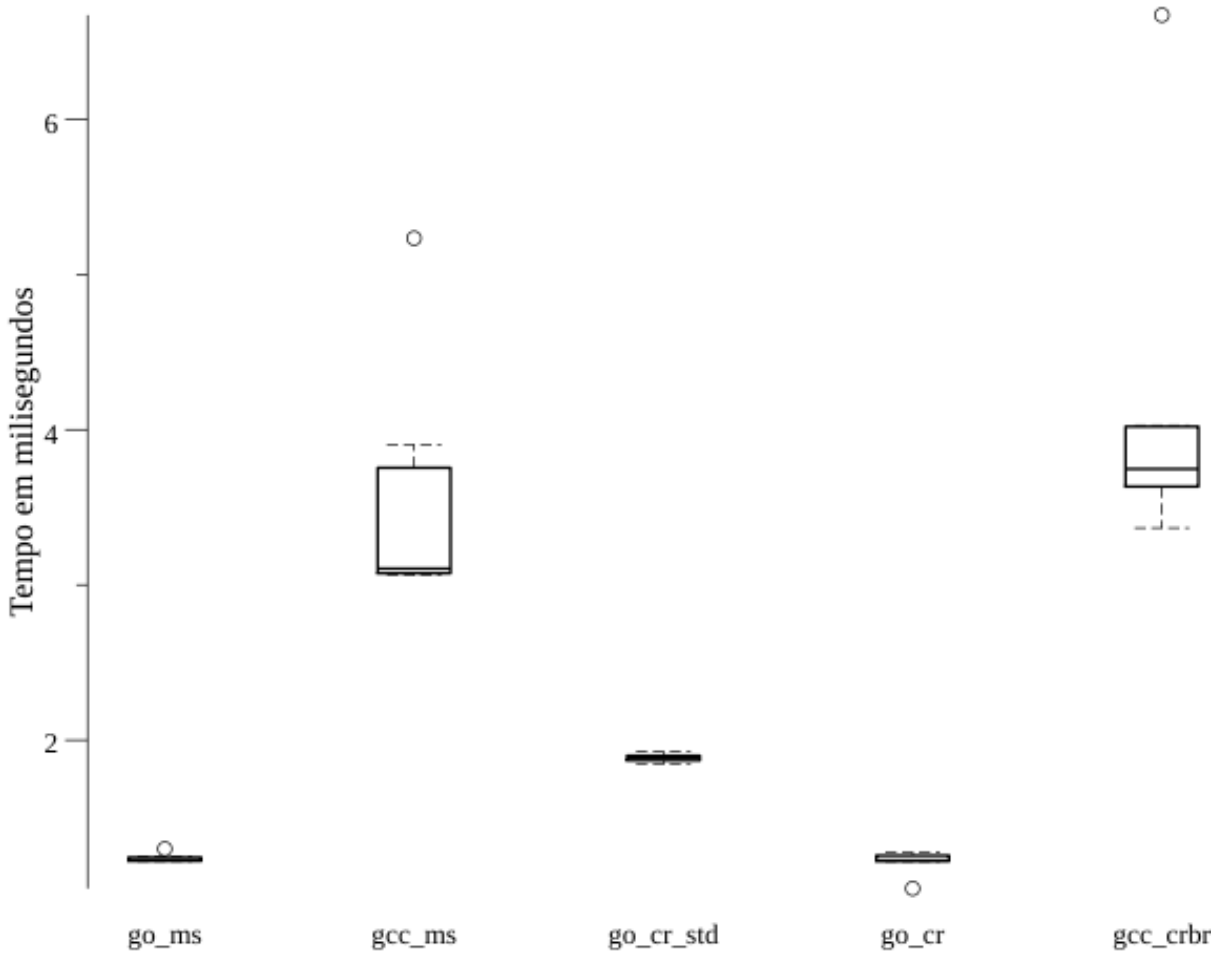
Tabela 10 – Resultados para *benchmark* Matmul na Máquina 2.

| **Entrada 100** | | |
|---|---|---|
| **Coletor** | go_ms | go_cr |
| **Tempo de execução** | 1,05$ms$ | 1,07$ms$ |
| **Mem alocada** | 0,43$MB$ | 0,43$MB$ |
| **Mem liberada** | $\approx 0\%$ | $\approx 99\%$ |
| **Número de pausas** | 0 | 0 |
| **Desvio padrão tm** | 0,014 | 0,005 |
| **Desvio padrão Mem** | 22,07 | 19,52 |
| | 0,00 0,00 | |
| ***p-value* Tempo/Mem** | 0,00 0,00 | |
| | 0,99 1 | |
| **Entrada 1000** | | |
| **Tempo de execução** | 857,90$s$ | 874,14$s$ |
| **Mem alocada** | 32,94$MB$ | 32,93$MB$ |
| **Mem liberada** | $\approx 75,62\%$ | $\approx 99\%$ |
| **Número de pausas** | 3 | 0 |
| **Desvio padrão tm** | 2,23 | 1,31 |
| **Desvio padrão Mem** | 270,83 | 31,83 |
| | 0,00 1 | |
| ***p-value* Tempo/Mem** | 0,00 0,00 | |
| | 1 0,00 | |
| Entrada 5000 | | |
| **Tempo de execução** | 117,88$s$ | 118$s$ |
| **Mem alocada** | 819,77$MB$ | 819,76$MB$ |
| **Mem liberada** | $\approx 99,9\%$ | $\approx 99,9\%$ |
| **Número de pausas** | 8 | 0 |
| **Desvio padrão tm** | 0,29 | 0,14 |
| **Desvio padrão Mem** | 274,45 | 23,83 |
| | 0,19 1 | |
| ***p-value* Tempo/Mem** | 0,38 0,00 | |
| | 0,82 0,00 | |

#### 5.4.3.3 Máquina 3

A última arquitetura utilizada provê maior quantidade de memória disponível, e sendo que esse experimento não realiza uma grande quantidade de alocações, observa-se que o tempo e processamento caem quando comparado às outras arquiteturas com menor quantitativo de memória disponível. A Figuras 23, 24 e 25 ilustram esses resultados, porém, o comportamento observado anteriormente se repete, logo, pode-se ver que os coletores em Go ainda apresentam resultados semelhantes, porém com uma variação entre as abordagens do algoritmo original e a versão proposta nesta tese, enquanto neste último caso o coletor por CR baseado em regiões de memória levou mais tempo para finalizar toda a carga do teste.

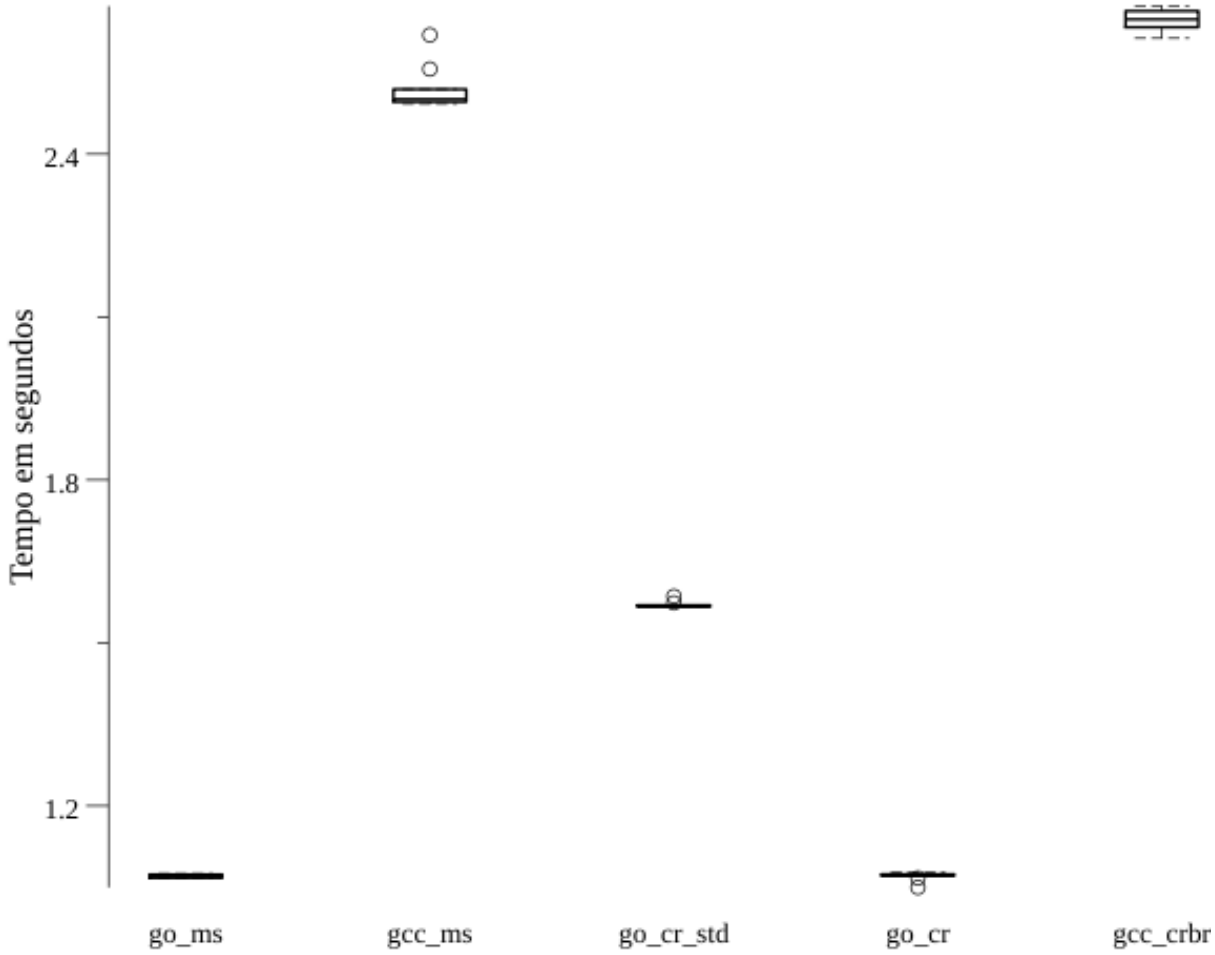
Figura 23 – Matmul Máquina 3 com entrada igual a 100.




**Fonte:** o autor.

Assim como nas outras arquiteturas, não foi necessária nenhuma coleta com a primeira entrada do programa para o compilador em Go com a coleta MS, como observado na Tabela 11. Aqui, podemos observar tanto o valor da média como nos resultados estatísticos

Figura 24 – Matmul Máquina 3 com entrada igual a 1000.

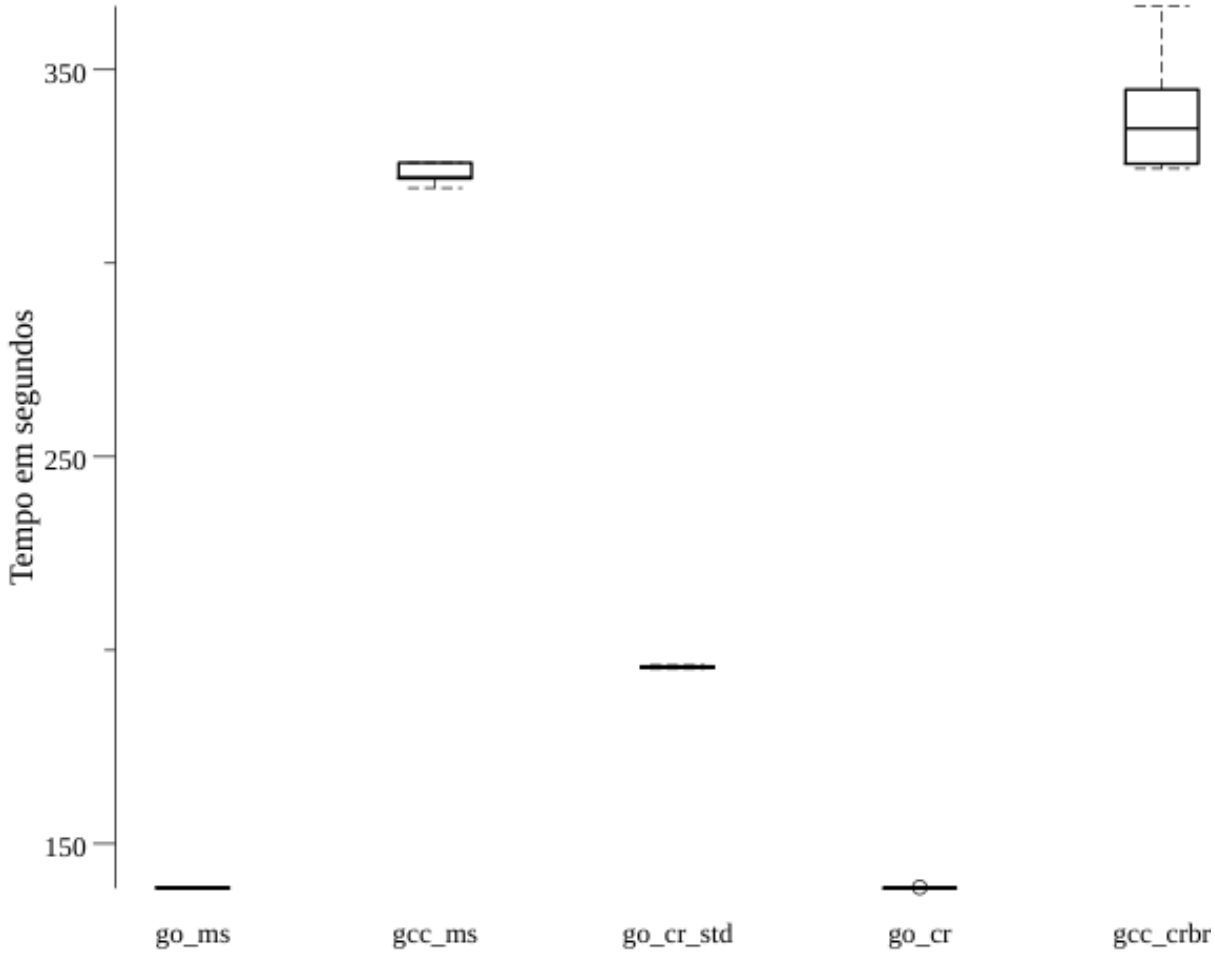



**Fonte:** o autor.

que a coleta por CR é mais eficiente para o tempo de execução, mas não no consumo de memória, e ela também desempenhou quase 99% da coleta dos dados ao longo da execução.

Dada a segunda entrada do teste (1000), o número de coleta aumenta por parte do coletor por MS, que totaliza 3 pausas completas do programa, enquanto a coleta por CR não necessita de nenhuma pausa. Os dados estatísticos mais uma vez elucidam as informações. Pode-se ver que os valores não são tão diferentes, mas, ainda assim, apresentam diferentes distribuições, sendo que a coleta por CR é menor em tempo de execução estatisticamente, porém, consome maior memória, dados os valores calculados.

Figura 25 – Matmul Máquina 3 com entrada igual a 5000.




**Fonte:** o autor.

Tabela 11 – Resultados para *benchmark* Matmul na Máquina 3.

| **Entrada 100** | | | |
|---|---|---|---|
| **Coletor** | go_ms | | go_cr |
| **Tempo de execução** | 1,24$ms$ | | 1,22$ms$ |
| **Mem alocada** | 0,43$MB$ | | 0,43$MB$ |
| **Mem liberada** | $\approx 0,0\%$ | | $\approx 96,0\%$ |
| **Número de pausas** | 0 | | 0 |
| **Desvio padrão tm** | 0,024 | | 0,06 |
| **Desvio padrão Mem** | 191 | | 109,39 |
| | 0,71 | 0,00 | |
| ***p-value* Tempo/Mem** | 0,63 | 0,01 | |
| | 0,31 | 0,99 | |
| **Entrada 1000** | | | |
| **Tempo de execução** | 1,07$s$ | | 1,07$s$ |
| **Mem alocada** | 32,94$MB$ | | 32,93$MB$ |
| **Mem liberada** | $\approx 75,64\%$ | | $\approx 99,95\%$ |
| **Número de pausas** | 3 | | 0 |
| **Desvio padrão tm** | 0,003 | | 0.007 |
| **Desvio padrão Mem** | 161,03 | | 54,43 |
| | 0,13 | 1 | |
| ***p-value* Tempo/Mem** | 0,27 | 0,00 | |
| | 0,88 | 0,00 | |
| **Entrada 5000** | | | |
| **Tempo de execução** | 138,53$s$ | | 138,52$s$ |
| **Mem alocada** | 819,77$MB$ | | 819,76$MB$ |
| **Mem liberada** | $\approx 75\%$ | | $\approx 99,9\%$ |
| **Número de pausas** | 9 | | 0 |
| **Desvio padrão tm** | 0,04 | | 0,06 |
| **Desvio padrão Mem** | 151,1 | | 3663,25 |
| | 0,76 | 0,99 | |
| ***p-value* Tempo/Mem** | 0,49 | 0,00 | |
| | 0,25 | 0,00 | |

Na última entrada, confirma-se que os valores da coleta por CR é mais eficiente no final da execução do teste, tanto no que diz respeito ao tempo de execução como em consumo de memória, além de não precisar suspender a execução do programa para coleta de dados, e coletando expressivamente mais objetos na *heap*, quando comparado a coleta por MS.

### 5.4.4 Garbage

O *benchmark* Garbage foi o experimento dentre os demais presentes nessa tese que exigiu maior quantidade do total de memória alocada. Mesmo não sendo esse valor alocado em um único passo, isso gerou uma grande carga para todas as arquiteturas. Somente a máquina com maior quantidade de memória (Máquina 3 com $16GB$ de memória RAM) foi capaz de executar esse teste. Por essa razão, os valores apresentados a seguir são unicamente relacionados a essa arquitetura.

Como pode ser observado na Tabela 12, a coleta por CR apresentou menor tempo de execução, tanto por média como nos valores estatísticos. Dado o alto valor de memória utilizada, foi necessário o disparo de varreduras locais para possíveis coletas de objetos em ciclo totalizando 7 pausas para liberação de memória.

Tabela 12 – Resultados para *benchmark* Garbage na Máquina 3.

| **Coletor** | go_ms | go_cr |
|---|---|---|
| **Tempo de execução** | 32,08$s$ | 23,45$s$ |
| **Memória alocada** | 132,25$G$ | 132,24$G$ |
| **Número de pausas** | 48 | 7 |
| **Memória liberada** | $\approx 7,6\%$ | $\approx 76,45\%$ |

O teste alocou em geral mais de $130GB$. A coleta por MS realizou um total de 48 pausas e liberou 7,5% do total da memória alocada, enquanto a coleta por CR apresentou maior eficiência de tempo de execução e total de memória utilizada. Os valores apresentados em ambas as Tabelas 13 e 14 confirmam o antes observado nos valores das médias.

Tabela 13 – Resultados para *benchmark* Garbage $p - values$ para tempo de execução.

| **Hipótese** | ***p-value*** | **Desvio padrão** | |
|---|---|---|---|
| | | **go_ms** | **go_cr** |
| -1 | 1 | | |
| 0 | 0,00 | 0,14 | 0,07 |
| 1 | 0,00 | | |

Inicialmente, é útil investigar se existe diferença significativa entre as taxas, sob distintos cenários. Com essa finalidade, utilizou-se o teste de Mann-Whitney, o qual apresenta em sua hipótese nula que as medianas são iguais. É possível testar contra a alternativa

que é menor que, não é igual a, ou maior que, respectivamente. Esse teste compara o valor da mediana ao invés da média e não faz nenhuma suposição quanto à distribuição populacional.

Tabela 14 – Resultados para *benchmark* Garbage $p - values$ para memória.

| **Hipótese** | ***p-value*** | **Desvio padrão** | |
|---|---|---|---|
| | | **go_ms** | **go_cr** |
| -1 | 1 | | |
| 0 | 0,00 | 24065,0 | 20145,2 |
| 1 | 0,00 | | |

Não são apresentados valores dos demais coletores (gcc_ms, go_cr_std e gcc_crbr) porque a CR com o algoritmo original apresenta erro por memória insuficiente, enquanto os compiladores com coleta por MS e CR com base e regiões de memória não suportaram as especificidades do *benchmark*. Isso se deu pela versão do GCC utilizado, uma vez que os programas foram elaborados para novas versões dos compiladores de Go, os coletores não foram capazes de reconhecer as estruturas e especificações utilizadas para produção do teste, sendo necessária uma versão mais recente para ter suporte, sendo impossível gerar um código binário para o presente teste.

Com base no teste de hipóteses realizado, podemos observar, na Tabela 14, que em sua grande maioria, a Hipótese $H_0$ é rejeitada ao nível de 5% de significância. Dessa maneira, podemos concluir que existe diferença significativa entre as medianas dos testes analisados.

### 5.4.5 SmallHeap

O SmallHeap também foi utilizado para averiguação do desempenho do coletor Go. Com alta taxa de alocação, em pequenas porções de objetos alocados, o SmallHeap não faz uso de grande quantidade total de alocação, mas força o coletor a operar em diversas coletas por sobrecarregar subáreas com objetos de tamanhos iguais. A Tabela 15 apresenta o resultado do desempenho.

Mesmo com um valor expressivo de número de paradas, o total de memória reclamada foi baixa na coleta por MS. Isso pode se explicar pelo baixo número total de alocação, visto que ainda há espaço para se solicitar mais memória ao SO, o coletor opta por reduzir o tempo de suspensão de coleta, porém isso também pode causar um alto valor no número total de pausas.

A CR consegue reclamar muitos objetos ao longo do período de execução, dado o baixo montante de total de memória a busca por liberação de memória por varredura local é desnecessária, sendo que não foi feita nenhuma pausa nas execuções.

Tabela 15 – Resultados para *benchmark* SmallHeap.

| **Coletor** | **Tempo de execução** | **Mem. alocada** | **Mem. liberada** | **Nº de pausas** |
|---|---|---|---|---|
| **Máquina 1** | | | | |
| go_ms | 10,21$s$ | 51,13$MB$ | $\approx 0,29\%$ | 13 |
| go_cr | 10,15$s$ | 0,14$MB$ | $\approx 80,8\%$ | 0 |
| **Máquina 2** | | | | |
| go_ms | 10,17$s$ | 84,42$MB$ | $\approx 0,18\%$ | 16 |
| go_cr | 10,20$s$ | 152,27$KB$ | $\approx 88,7\%$ | 0 |
| **Máquina 3** | | | | |
| go_ms | 10,17$s$ | 82,82$MB$ | $\approx 0,23\%$ | 16 |
| go_cr | 10,12$s$ | 19,46$MB$ | $\approx 0,75\%$ | 0 |

O número de coletas apresentado não é significante, como no teste anterior, porém são observados em todas as arquiteturas a coleta por CR. A coleta por CR libera mais objetos que a outra abordagem e consegue ser, em média, mais eficiente em tempo de execução em alguns casos.

Tabela 16 – Resultados para *benchmark* SmallHeap, $p-values$ para o tempo de execução.

| **Hipótese** | ***p-value*** | **Desvio padrão** | |
|---|---|---|---|
| | | **go_ms** | **go_cr** |
| **Máquina 1** | | | |
| -1 | 1 | 0,002 | 0,003 |
| 0 | 0,00 | | |
| 1 | 0,00 | | |
| **Máquina 2** | | | |
| -1 | 0,00 | 0,0007 | 0,002 |
| 0 | 0,00 | | |
| 1 | 1 | | |
| **Máquina 3** | | | |
| -1 | 0,91 | 0,0008 | 0,001 |
| 0 | 0,19 | | |
| 1 | 0,09 | | |

Os dados estatísticos para o tempo de execução presentes na Tabela 16 mudam à medida que são executados os experimentos nas diferentes arquiteturas, principalmente devido a quantidade total de memória RAM disponível. Na primeira máquina, tem melhor desempenho que a CR se sai melhor que a coleta por MS, porém isso se inverte na segunda

máquina. Na terceira, a CR torna a desempenhar um resultado favorável em tempo de execução.

Estatisticamente, são demonstradas na Tabela 17 as hipóteses da coleta por CR e MS, onde pode-se verificar que a CR usou menos memória nas duas primeiras arquiteturas, enquanto na última máquina, com maior quantidade de memória disponível, os valores foram estatisticamente similares.

A Tabela 16 ilustra os resultados do teste não paramétrico de Mann-Whitney. Para a Máquina 1, ao nível de 5% de significância, a hipótese $H_0$ do teste bilateral é rejeitada, ou seja, existe diferença entre as medianas do tempo de execução para o benchmark SmallHeap. Além disso, existe evidência significativa de que a mediana do modelo go_ms é maior que o modelo go_cr. Nessa mesma Tabela, é possível observar que, sob o cenário da Máquina 2, a hipótese inicial do teste bilateral foi rejeitada ao nível nominal de 5%, indicando assim que existe diferença entre os valores analisados. Sob a Máquina 3, ao nível de 5% de significância, as hipóteses alternativas foram rejeitadas, ou seja, as medianas são iguais em relação ao tempo de execução do benchmark SmallHeap.

Os resultados corroboram com a descrição do experimento, com alta taxa de pequenas alocações a coleta por CR libera diversos objetos, reutilizando o mesmo espaço de memória sem dificuldades, enquanto a coleta por MS precisa realizar todo um ciclo de coleta para poder liberar um bloco inteiro de memória, custando muito mais tempo de pausa.

De acordo com o teste não-paramétrico abordado neste trabalho, os resultados para a memória sob diferentes hipóteses são apresentados na Tabela 17. Na máquina 1, observa-se que as medianas são diferentes e que a mediana do $go\_ms$ é maior que a mediana $go\_cr$, ao nível nominal de 5%. Com o mesmo nível de significância abordado, observa-se que não existe diferença entre o tempo de memória do benchmark SmallHeap para a Máquina 3, ou seja, a mediana de ambos os testes são iguais.

Tabela 17 – Resultados para *benchmark* SmallHeap $p - values$ para a memória.

| **Hipótese** | ***p-value*** | **Desvio padrão** | |
|---|---|---|---|
| | | **go_ms** | **go_cr** |
| **Máquina 1** | | | |
| -1 | 1 | 1,90 | 23,53 |
| 0 | 0,00 | | |
| 1 | 0,00 | | |
| **Máquina 2** | | | |
| -1 | 1 | 2,96 | 42,08 |
| 0 | 0,00 | | |
| 1 | 0,00 | | |
| **Máquina 3** | | | |
| -1 | 0,48 | 2,87 | 2,95 |
| 0 | 0,97 | | |
| 1 | 0,54 | | |

### 5.4.6 LargeBSS

O último experimento realiza poucas alocações, e isso impacta o número de coletas feitas pelo coletor com uso de MS em todas as arquiteturas testadas.

Conforme apresentado na Tabela 18, a coleta por CR realizou mais liberação de objetos da *heap*, enquanto a coleta por MS realizou poucos ciclos de coletas, isso foi observado em todas as arquiteturas. Esse resultado mostra o impacto do número de pausas na execução total do programa.

Tabela 18 – Resultados para *benchmark* LargeBSS.

| **Coletor** | **Tempo de execução** | **Mem. alocada** | **Mem. liberada** | **Nº de pausas** |
|---|---|---|---|---|
| **Máquina 1** | | | | |
| go_ms | 10,12$s$ | 4,75$MB$ | $\approx 0,01\%$ | 1 |
| go_cr | 13,91$s$ | 99,89$KB$ | $\approx 0,53\%$ | 0 |
| **Máquina 2** | | | | |
| go_ms | 10,06$s$ | 5,18$MB$ | $\approx 0,008\%$ | 1 |
| go_cr | 10,26$s$ | 117,3$KB$ | $\approx 0,49\%$ | 0 |
| **Máquina 3** | | | | |
| go_ms | 10,08$s$ | 10,70$MB$ | $\approx 0,003\%$ | 3 |
| go_cr | 10,39$s$ | 109,82$KB$ | $\approx 0,49\%$ | 0 |

Os valores das hipóteses na Tabela 19 mostram que a CR realizou a execução do

teste em maior tempo em todas as arquiteturas, enquanto a coleta por MS apresentou menores valores, o tempo de execução por MS não sofreu baixa no valor de execução por desempenhar um número baixo de ciclos de coleta, logo, o teste não sofreu grandes interrupções. Isso é confirmado ao verificar-se o valor de memória liberada ao longo da execução do experimento, o que foi observado em todas as arquiteturas usadas.

Além disso, podemos verificar na Tabela 19 o desempenho do teste não paramétrico utilizado. Independente dos tipos de máquinas utilizadas, o teste apresentou resultados similares para todos os cenários. Dessa maneira, ao nível nominal de 5%, rejeitamos a hipótese inicial, ou seja, existe diferença significativa entre as medianas. Também podemos concluir que o *go_ms* possui um valor significativamente menor.

Tabela 19 – Resultados para *benchmark* LargeBSS $p - values$ de Tempo de execução.

| **Hipótese** | ***p-value*** | **Desvio padrão** | |
|---|---|---|---|
| | | **go_ms** | **go_cr** |
| **Máquina 1** | | | |
| -1 | 0,00 | | |
| 0 | 0,00 | 0,0008 | 0,41 |
| 1 | 1 | | |
| **Máquina 2** | | | |
| -1 | 0,00 | | |
| 0 | 0,00 | 0,0007 | 0,002 |
| 1 | 1 | | |
| **Máquina 3** | | | |
| -1 | 0,00 | | |
| 0 | 0,00 | 0,0003 | 0.005 |
| 1 | 1 | | |

Tabela 20 – Resultados para *benchmark* LargeBSS $p - values$ de uso de memória.

| **Hipótese** | ***p-value*** | **Desvio padrão** | |
|---|---|---|---|
| | | **go_ms** | **go_cr** |
| **Máquina 1** | | | |
| -1 | 1 | 171,21 | 41,89 |
| 0 | 0,00 | | |
| 1 | 0,00 | | |
| **Máquina 2** | | | |
| -1 | 1 | 274,45 | 23,83 |
| 0 | 0,00 | | |
| 1 | 0,00 | | |
| **Máquina 3** | | | |
| -1 | 1 | 184,83 | 43,21 |
| 0 | 0,00 | | |
| 1 | 0,00 | | |

Já os valores, no que diz respeito ao consumo de memória apresentados na Tabela 20, evidencia-se que a coleta por CR é mais eficiente em consumo de memória, por utilizar menos memória para desempenhar o mesmo experimento e conseguir desalocar mais objetos da *heap*, e não foi necessário nenhuma pausa por meio da coleta por CR.

# 6 CONSIDERAÇÕES FINAIS

Essa tese apresentou estudo sobre o gerenciamento automático de memória, propondo nova abordagem envolvendo os algoritmos de contagem de referências com detecção de estruturas cíclicas e regiões de memória. Durante a implementação, foram feitas otimizações essenciais nas técnicas da literatura para garantir o melhoramento na detecção e na integração com a abordagem proposta. Além disso, os resultados foram comprovados por experimentação.

Na abordagem proposta, as otimizações desenvolvidas estão ligadas ao uso mútuo das técnicas de contagem de referências com o gerenciamento baseado em regiões de memória, objetivando maximizar o controle do uso de memória de forma dinâmica. Este trabalho tem como principal objetivo otimizar o sistema de coleta de dados, reduzindo o número de suspensões dos programas por meio da contagem de referências em regiões de memória.

O Capítulo 4 apresentou as propostas de otimizações para as técnicas apresentadas, de modo a maximizar o desempenho do sistema de coleta, buscando minimizar os efeitos causados pela coleta de dados durante a execução do programa.

O Capítulo 5 sintetizou os principais experimentos juntamente com a análise de benchmarks em diversas arquiteturas. Demonstrou-se que o sistema de coleta de dados com contagem de referências com detecção de ciclos reduziu significativamente o número de pausas realizadas, em comparação com o algoritmo de contagem de referências e detecção de ciclos, observam-se ganhos de desempenho em até 30% em tempo de execução, com os coletores incrementais e melhoramentos no sistema de alocamento. Além disso, foi possível otimizar o consumo total de memória em 50% a menos de uso de memória para programas que exigem grandes quantidades de memória e não fazem pausas no processo do usuário. Esses valores podem apresentar maiores ganhos quando combinadas as técnicas de contagem de referências e o gerenciamento de memória baseado em regiões.

A técnica de CR, quando comparada à abordagem de MS, é mais eficiente quando o número de coletas é maior, pois o número de ciclos de coletas se estende, forçando a elevado número de pausas no programa. Isso impacta o tempo de execução dos sistemas.

Constatou-se que mesmo os programas mais simples sempre terão maior número de coleta quando usada a abordagem por CR. Isso é justificável pela realização das coletas concomitantemente à execução dos programas. Por essa razão, a abordagem CR consumiu tempo maior de execução em alguns cenários quando comparada à coleta por MS, por não apresentar poucos ou nenhum ciclo de coleta, porém em outras ocasiões os resultados se invertem, mostrando que, à medida em que o número de pausas aumenta na coleta por MS e juntamente a isso número de pausas também aumenta, a coleta por CR pode apresentar menor tempo de execução por inferir menor tempo de pausa.

Assim, é possível responder as perguntas expostas no início desta pesquisa. A primeira

Questão de Pesquisa (QP) indaga sobre o impacto no tempo de execução com a introdução de pausas para coleta por CR para detecção de estruturas em ciclos. Baseado nos resultados apresentados no Capítulo 5, pôde-se notar que o tempo de execução pode ser reduzido, isso é observado quando comparamos o uso do coletor apresentado por Formiga (2011) e a CR com as otimizações apresentadas na tese, quando o número de coletas cresce, o tempo de execução sofre o impacto direto, dada a atuação do coletor, porém foi observado que esse impacto pode ser minimizado de duas maneiras propostas na tese: primeiro o impacto das pausas pode ser reduzido limitando o tempo de atuação da detecção de estruturas em ciclo,reduzindo-se a varredura local para atuar em uma região específica da memória, aqui chamada de lista segregada por tamanho, ou seja, a varredura que já era feita de forma local passou a ter menor área de atuação com maior potencial de identificação de células que podem ser liberadas, de modo que esse espaço a ser liberado seja compatível com a necessidade do objeto a ser alocado. A segunda maneira de reduzir o impacto das pausas para detecção de objetos em ciclo foi usando o gerenciamento de memória por região, novamente limitando a atuação da varredura local em um único segmento/região da memória.

A partir dos experimentos, também é possível afirmar a possibilidade de reduzir o número de pausas em programas que usam gerenciamento de memória dinâmica, que responde a segunda QP: é possível eliminar ou reduzir o número de pausas para ação da coleta de lixo? As varreduras locais para detecção de estruturas em ciclos podem reduzir o número de pausas total para coleta de lixo, além disso a junção da CR com o gerenciamento de memória por regiões também reduz o número de pausas realizadas, porque o gerenciamento dinâmico é preestabelecido durante a compilação do programa.

A terceira QP é relacionada ao desempenho no tempo de execução e principalmente no total de memória utilizada quando a redução das operações nos contadores em objetos na heap, diferindo do uso padrão da CR, bem como reduzir a quantidade de uso de memória por otimizações na forma como os dados são alocados. Essa redução pode diminuir o total de memória consumida. Isso pode ser observado principalmente quando comparamos os resultados de CR original apresentada por Formiga (2011) e a versão proposta na tese. Outros trabalhos na literatura já haviam ilustrado que é possível retirar a atomicidade da operação dos contadores para as referências de alguns objetos Choi, Shull e Torrellas (2018). Nesta tese, entretanto, propomos que objetos alocados na heap (exemplo variáveis constantes) não precisam de contadores, porque esses objetos serão mantidos na *heap* durante toda a execução do programa. Com isso é possível reduzir o número de contadores no heap, reduzindo consequentemente o espaço total de memória utilizado e operações adicionais durante o tempo de execução.

A última **QP** pergunta refere-se a se é possível reduzir o número de varreduras na detecção de estruturas cíclicas mantendo-se o desempenho da contagem de referências. Mostra-se que, com o uso da alocação por lista livre e segregada, as varreduras locais

podem ser subdivididas em varreduras menores, assim como o gerenciamento de memória por regiões pode reduzir a ação da coleta por CR, reduzindo, consequentemente, a coleta e as varreduras locais.

Assim as melhorias e a junção do uso das técnicas de contagem de referência com o gerenciamento baseado em regiões de memória, proposto no Capítulo 4, podem ser usadas com vantagem em sistemas de tempo real, em que o programa deve responder às solicitações em prazos limitados, bem como serviços Web e outros, em que o tempo total de processamento é importante.

Ameaças à validade

Seguimos os critérios da taxonomia apresentada por Wohlin et al. (2012) para classificar as ameaças à validade do trabalho, como ameaças à validade interna e externa, ameaças de constructo e de conclusão. Apresentamos como ameaça à validade interna o fato de que não foi possível replicar as condições de uso para coleta com o compilador criado por Davis (2015). Isso impossibilitou a comparação e análise do coletor por região de memória e coleta por cópia originalmente presentes no trabalho.

Outra ameaça à validade externa do trabalho se dá pelo uso exclusivo de duas técnicas principais: a contagem de referências e marcação e varredura, bem como a limitação do uso das mesmas em linguagem única. Além disso, é interessante buscar outros experimentos com natureza e complexidades diferenciadas, permitindo a consolidação dos resultados aqui apresentados.

Os demais critérios de ameaças como constructo e ameaça à validade de conclusão são sanadas no trabalho com o uso da análise estatística para comprovar a afirmação dos dados obtidos nas experimentações e o uso de benchmarks de terceiros para mitigar eventual viés da avaliação das técnicas e melhorias propostas.

## 6.1 AMEAÇAS À VALIDADE

Seguimos os critérios da taxonomia apresentada por Wohlin et al. (2012) para classificar as ameaças à validade do trabalho, como ameaças à validade interna e externa, ameaças de constructo e de conclusão. Apresentamos como ameaça à validade interna o fato de que não foi possível replicar as condições de uso para coleta com o compilador criado por Davis (2015). Isso impossibilitou a comparação e análise do coletor por região de memória e coleta por cópia originalmente presentes no trabalho.

Outra ameaça à validade externa do trabalho se dá pelo uso exclusivo de duas técnicas principais: a contagem de referências e marcação e varredura, bem como a limitação do uso das mesmas em linguagem única. Além disso, é interessante buscar outros experimentos com natureza e complexidades diferenciadas, permitindo a consolidação dos resultados aqui apresentados.

Os demais critérios de ameaças como constructo e ameaça à validade de conclusão são sanadas no trabalho com o uso da análise estatística para comprovar a afirmação

dos dados obtidos nas experimentações e o uso de *benchmarks* de terceiros para mitigar eventual viés da avaliação das técnicas e melhorias propostas.

## 6.2 TRABALHOS FUTUROS

O presente trabalho investigou aspectos relevantes para a área da computação que analisa a coleta de lixo. Por fim, propomos as expansões a seguir como trabalhos futuros ao exposto nesta tese:

- Explorar o gerenciamento de memória aqui apresentado com número maior de coletores e mutadores;
- Explorar compiladores inteligentes em que se possa definir, em tempo de compilação, qual a coleta mais adequada a ser utilizada a partir da análise da compilação.

# REFERÊNCIAS


ANDREW, W. A. *Modern Compiler Implementation in ML*. [S.l.]: Cambridge University Press, 1998.

APPEL, A. W. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, Wiley Online Library, v. 19, n. 2, p. 171–183, 1989.

AZATCHI, H.; PETRANK, E. Integrating generations with advanced reference counting garbage collectors. In: SPRINGER. *International Conference on Compiler Construction*. [S.l.], 2003. p. 185–199.

BACKUS, J. W. Automatic programming: properties and performance of fortran systems i and ii. In: *Proceedings of the Symposium on the Mechanisation of Thought Processes*. [S.l.: s.n.], 1958. p. 165–180.

BACON, D. F.; ATTANASIO, C. R.; RAJAN, V.; SMITH, S. E.; LEE, H. A pure reference counting garbage collector. *ACM*, 2001.

BOETTIGER, C. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, ACM, v. 49, n. 1, p. 71–79, 2015.

CHOI, J.; SHULL, T.; TORRELLAS, J. Biased reference counting: minimizing atomic operations in garbage collection. In: ACM. *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. [S.l.], 2018. p. 35.

CLINGER, W. D.; HANSEN, L. T. Generational garbage collection and the radioactive decay model. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1997. v. 32, n. 5, p. 97–108.

COLLINS, G. E. A method for overlapping and erasure of lists. *Communications of the ACM*, ACM, v. 3, n. 12, p. 655–657, 1960.

DAVIS, M. *Automatic memory management techniques for the go programming language.* Tese (Doutorado) — University of Melbourne, Australia, 2015.

DAVIS, M.; SCHACHTE, P.; SOMOGYI, Z.; SØNDERGAARD, H. Towards region-based memory management for go. In: ACM. *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. [S.l.], 2012. p. 58–67.

DEGENBAEV, U.; EISINGER, J.; ERNST, M.; MCILROY, R.; PAYER, H. Idle time garbage collection scheduling. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2016. v. 51, n. 6, p. 570–583.

DESHPANDE, N.; SPONSLER, E.; WEISS, N. Analysis of the go runtime scheduler. *URl: http://www. cs. columbia. edu/˜ aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO. pdf (visited on 2016-12-19)*, 2012.

DIJKSTRA, E. W.; LAMPORT, L.; MARTIN, A. J.; SCHOLTEN, C. S.; STEFFENS, E. F. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, ACM, v. 21, n. 11, p. 966–975, 1978.

DOCKER-HUB. *Docker*. 2019. Disponível em: <https://www.docker.com/>.

DOMANI, T.; KOLODNER, E. K.; LEWIS, E.; SALANT, E. E.; BARABASH, K.; LAHAN, I.; LEVANONI, Y.; PETRANK, E.; YANORER, I. Implementing an on-the-fly garbage collector for java. *ACM SIGPLAN Notices*, ACM, v. 36, n. 1, p. 155–166, 2001.

DONOVAN, A. A.; KERNIGHAN, B. W. *The Go programming language*. [S.l.]: Addison-Wesley Professional, 2015.

FENICHEL, R. R.; YOCHELSON, J. C. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, ACM, v. 12, n. 11, p. 611–612, 1969.

FÉREY, G.; SHANKAR, N. Code generation using a formal model of reference counting. In: SPRINGER. *NASA Formal Methods Symposium*. [S.l.], 2016. p. 150–165.

FERREIRO, H.; CASTRO, L.; JANJIC, V.; HAMMOND, K. Kindergarten cop: dynamic nursery resizing for ghc. In: ACM. *Proceedings of the 25th International Conference on Compiler Construction*. [S.l.], 2016. p. 56–66.

FLOOD, C. H.; DETLEFS, D.; SHAVIT, N.; ZHANG, X. Parallel garbage collection for shared memory multiprocessors. In: *Java Virtual Machine Research and Technology Symposium*. [S.l.: s.n.], 2001.

FORMIGA, A. d. A. *Algoritmos para contagem de referências cíclicas em sistemas multiprocessados*. Tese (Doutorado) — Universidade Federal de Pernambuco, 2011.

FORMIGA, A. de A.; LINS, R. D. A new architecture for concurrent lazy cyclic reference counting on multi-processor systems. *J. UCS*, v. 13, n. 6, p. 817–829, 2007.

FULGHAM, B.; GOUY, I. *The computer language benchmarks game*. 2009. Disponível em: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>.

GARNER, R.; BLACKBURN, S. M.; FRAMPTON, D. Effective prefetch for mark-sweep garbage collection. In: ACM. *Proceedings of the 6th international symposium on Memory management*. [S.l.], 2007. p. 43–54.

GARNER, R. J.; BLACKBURN, S. M.; FRAMPTON, D. A comprehensive evaluation of object scanning techniques. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2011. v. 46, n. 11, p. 33–42.

GAY, D.; AIKEN, A. *Memory management with explicit regions*. [S.l.]: ACM, 1998. v. 33.

GIDRA, L.; THOMAS, G.; SOPENA, J.; SHAPIRO, M. A study of the scalability of stop-the-world garbage collectors on multicores. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2013. v. 48, n. 4, p. 229–240.

GOLANG. *Golang Research Papers*. 2019. Disponível em: <https://github.com/golang/go/wiki/ResearchPapers>.

GROSSMAN, D.; MORRISETT, G.; JIM, T.; HICKS, M.; WANG, Y.; CHENEY, J. Region-based memory management in cyclone. *ACM Sigplan Notices*, ACM, v. 37, n. 5, p. 282–293, 2002.

HARVEY-LEES-GREEN, N.; BIGLARI-ABHARI, M.; MALIK, A.; SALCIC, Z. A dynamic memory management unit for real time systems. In: IEEE. *Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on*. [S.l.], 2017. p. 84–91.

HELLYER, L.; JONES, R.; HOSKING, A. L. The locality of concurrent write barriers. *ACM Sigplan Notices*, ACM, v. 45, n. 8, p. 83–92, 2010.

HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, ago. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359576.359585>.

HODGES, A. *Alan Turing: the enigma*. New York: Simon and Schuster, 1983. ISBN 978-0-671-49207-6 978-0-671-52809-6.

HUDSON, R. *Go GC: Prioritizing low latency and simplicity*. 2015. Disponível em: <https://blog.golang.org/go15gc>.

ILIEV, A.; KYURKCHIEV, N. V. 80th anniversary of the birth of prof. donald knuth (1938). *Biomath Communications*, v. 5, n. 1, 2018.

JAFFE, M. S.; LEVESON, N. G.; HEIMDAHL, M. P. E.; MELHART, B. E. Software requirements analysis for real-time process-control systems. *IEEE transactions on software engineering*, IEEE, n. 3, p. 241–258, 1991.

JONES, R.; LINS, R. D. Garbage collection: algorithms for automatic dynamic memory management. Wiley, 1996.

JONES, R. E.; LINS, R. D. Cyclic weighted reference counting without delay. In: SPRINGER. *International Conference on Parallel Architectures and Languages Europe*. [S.l.], 1993. p. 712–715.

KLOTZ, J. H. The wilcoxon, ties, and the computer. *Journal of the American Statistical Association*, Taylor & Francis Group, v. 61, n. 315, p. 772–787, 1966.

KOTZMANN, T.; MÖSSENBÖCK, H. Escape analysis in the context of dynamic compilation and deoptimization. In: ACM. *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. [S.l.], 2005. p. 111–120.

LANDIN, P. J. The next 700 programming languages. *Communications of the ACM*, ACM, v. 9, n. 3, p. 157–166, 1966.

LANG, B.; QUEINNEC, C.; PIQUER, J. Garbage collecting the world. In: ACM. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. [S.l.], 1992. p. 39–50.

LI, H.; WU, M.; CHEN, H. Analysis and optimizations of java full garbage collection. In: ACM. *Proceedings of the 9th Asia-Pacific Workshop on Systems*. [S.l.], 2018. p. 18.

LINS, R. D. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, Elsevier Science Bv, Po Box 211, 1000 Ae Amsterdam, Netherlands, v. 44, n. 4, p. 215–220, 1992.

LINS, R. D. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and microprogramming*, Elsevier, v. 35, n. 1-5, p. 563–568, 1992.

LINS, R. D. An efficient algorithm for cyclic reference counting. *Information Processing Letters*, Elsevier North-Holland, Inc., v. 83, n. 3, p. 145–150, 2002.

LINS, R. D. Cyclic reference counting. *Information Processing Letters*, Elsevier, v. 109, n. 1, p. 71–78, 2008.

LINS, R. D.; JONES, R. E. Cyclic weighted reference counting. In: *WP & FP'93 Workshop on Parallel and Distributed Processing, Sofia, Bulgaria*. [S.l.: s.n.], 1993. p. 369–382.

MACKINSON, T. N. Cobol: a sample problem. *Communications of the ACM*, ACM, v. 4, n. 8, p. 340–346, 1961.

MARLOW, S.; HARRIS, T.; JAMES, R. P.; JONES, S. P. Parallel generational-copying garbage collection with a block-structured heap. In: ACM. *Proceedings of the 7th international symposium on Memory management*. [S.l.], 2008. p. 11–20.

MARTIE, L.; PALEPU, V. K.; SAJNANI, H.; LOPES, C. Trendy bugs: Topic trends in the android bug reports. In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2012. (MSR '12), p. 120–123. ISBN 978-1-4673-1761-0. Disponível em: <http://dl.acm.org/citation.cfm?id=2664446.2664464>.

MARTINEZ, A.; WACHENCHAUZER, R.; LINS, R. *Cyclic reference counting with local mark-scan, IPL 34 (1990) 31—35*. [S.l.]: North Holland, 1990.

MCBETH, J. H. Letters to the editor: On the reference counter method. *Commun. ACM*, ACM, New York, NY, USA, v. 6, n. 9, p. 575–, set. 1963. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/367593.367649>.

MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, ACM, v. 3, n. 4, p. 184–195, 1960.

MCKNIGHT, P. E.; NAJAB, J. Mann-whitney u test. *The Corsini encyclopedia of psychology*, Wiley Online Library, p. 1–1, 2010.

MCLUHAN, M. *INTERNET GROWTH STATISTICS*. 2019. Disponível em: <https://www.internetworldstats.com/emarketing.htm>.

MINSKY, M. L. A lisp garbage collector algorithm using serial secondary storage. MIT Artificial Intelligence Project, Memo 58, 1963.

NANZ, S.; FURIA, C. A. A comparative study of programming languages in rosetta code. *CoRR*, abs/1409.0252, 2014. Disponível em: <http://arxiv.org/abs/1409.0252>.

PE-THAN, E. P. P.; DABBISH, L.; HERBSLEB, J. D. Collaborative writing on github: A case study of a book project. In: ACM. *Companion of the 2018 ACM Conference on Computer Supported Cooperative Work and Social Computing*. [S.l.], 2018. p. 305–308.

PERLIS, A. J.; SAMELSON, K. Preliminary report: International algebraic language. *Commun. ACM*, ACM, New York, NY, USA, v. 1, n. 12, p. 8–22, dez. 1958. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/377924.594925>.

PIKE, R. The go programming language. *Talk given at Google's Tech Talks*, 2009.

PIRINEN, P. P. Barrier techniques for incremental tracing. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1998. v. 34, n. 3, p. 20–25.

PIZLO, F.; ZIAREK, L.; MAJ, P.; HOSKING, A. L.; BLANTON, E.; VITEK, J. Schism: fragmentation-tolerant real-time garbage collection. In: ACM. *ACM Sigplan Notices*. [S.l.], 2010. v. 45, n. 6, p. 146–159.

RAD, B. B.; BHATTI, H. J.; AHMADI, M. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, International Journal of Computer Science and Network Security, v. 17, n. 3, p. 228, 2017.

REDMONK. *The RedMonk programming language rankings: Agosto 2018.* 2018. Disponível em: <https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/>.

REIJERS, N.; ELLUL, J.; SHIH, C.-S. Making sensor node virtual machines work for real-world applications. *IEEE Embedded Systems Letters*, IEEE, 2018.

REN, Y.; LIU, G.; PARMER, G.; BRANDENBURG, B. Scalable memory reclamation for multi-core, real-time systems. In: IEEE. *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. [S.l.], 2018. p. 152–163.

RICE, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, JSTOR, v. 74, n. 2, p. 358–366, 1953.

ROSETTA. *Rosetta Code*. 2019. Disponível em: <http://rosettacode.org/>.

ROVNER, P. *On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language*. [S.l.], 1985.

SAMMET, J. E. J. E. *Programming Languages; History and Fundamentals*. [S.l.], 1969.

SEWELL, W. *Golang's Real-time GC in Theory and Practice*. 2019. Disponível em: <https://making.pusher.com/golangs-real-time-gc-in-theory-and-practice/>.

SHAHRIYAR, R.; BLACKBURN, S. M.; FRAMPTON, D. Down for the count? getting reference counting back in the ring. *ACM SIGPLAN Notices*, ACM, v. 47, n. 11, p. 73–84, 2013.

TAHBOUB, R. *VLSI circuit partitioning for simulation and placement*. Tese (Doutorado) — Bilkent University, 1993.

TEREI, D.; LEVY, A. Blade: A data center garbage collector. *arXiv preprint arXiv:1504.02578*, 2015.

TIOBE. Tiobe programming community index. 2019.

UGAWA, T.; IWASAKI, H.; YUASA, T. Improved replication-based incremental garbage collection for embedded systems. In: ACM. *ACM Sigplan Notices*. [S.l.], 2010. v. 45, n. 8, p. 73–82.

UGAWA, T.; JONES, R. E.; RITSON, C. G. Reference object processing in on-the-fly garbage collection. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2014. v. 49, n. 11, p. 59–69.

VALKOV, I.; CHECHINA, N.; TRINDER, P. Comparing languages for engineering server software: Erlang, go, and scala with akka. ACM, 2018.

VECHEV, M. T.; BACON, D. F. Write barrier elision for concurrent garbage collectors. In: ACM. *Proceedings of the 4th international symposium on Memory management*. [S.l.], 2004. p. 13–24.

WASSERMAN, L. *All of statistics: a concise course in statistical inference*. [S.l.]: Springer Science & Business Media, 2013.

WELLINGS, A. J. *Concurrent and real-time programming in Java*. [S.l.]: John Wiley New York, 2004.

WILSON, P. R. Uniprocessor garbage collection techniques. In: *Memory Management*. [S.l.]: Springer, 1992. p. 1–42.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012.

YOUNIS, M. F. Memory allocation technique for segregated free list based on genetic algorithm. *Journal of Al-Nahrain University-Science*, Al-Nahrain University, v. 15, n. 2, p. 161–168, 2012.

# APÊNDICE A – GOLANG

## A.1 INTRODUÇÃO A GO

Golang tem proposta de ser uma linguagem para diversos tipos de uso, e a mesma já vem sendo usada em servidores web, jogos, sistemas de IoT, dentre outros.

A linguagem Go se destaca dentre tantas outras, possuindo as seguintes características:

- Linguagem imperativa (procedural e estrutural) construída com foco em concorrência, não dá suporte a objetos. Possui interfaces e funções de alta ordem;
- É estaticamente tipada e compila para código nativo, isso proporciona alto desempenho. Oferece suporte a tipos dinâmicos via determinação de tipos automaticamente para interfaces;
- É possível fazer *cross-compilation* e fornece suporte a diversos tipos de plataformas, apresentando algumas características de linguagens funcionais como *closure* e estruturas funcionais, como *maps* e *generics*;

A linguagem organiza os programas em pacotes (*package*), que são definidos no início do programa. As bibliotecas utilizadas são importadas de maneira a restringir o uso dos pacotes em um programa. Abaixo, ilustramos um código simples na linguagem Go.

```
package main

func main() {
    fmt.Println("Hello, world!")
}
```

Código A.1 – Código exemplo Hello World.

O exemplo de código acima pode ser executado de duas maneiras, usando o comando direto **go run hello_world.go** ou gerando o código executável com o comando **go build hello_world.go**. Após isso, o programa pode ser executado com **./hello_world**.

A linguagem permite dois estilos de concorrência, usando *goroutines* e *channels*, que são inspirados em Hoare (1978), no modelo de comunicação sequencial de processos, onde os valores são passados entre atividades independentes (*goroutines*) e como parte do pacote de processos assíncronos. Go usa *mutex* e tradicionais métodos de modelos baseados em memória compartilhada.

Com processamento de *threads* leves por meio das chamadas *goroutines*, Go combina eficiência, desempenho e segurança de uma linguagem fortemente tipada. Um dos objetivos propostos pela linguagem é tirar proveito da tendência de uso de multi-núcleos e arquiteturas distribuídas com suporte à redes, concorrência e paralelismo. Go atingiu o desempenho de execução de linguagens como C/C++, proporcionando o gerenciamento de memória dinamicamente.

## A.2 DECLARAÇÕES E ATRIBUIÇÕES

A sintaxe de declaração de variável e função é semelhante a C, mas com a nomes de tipo e identificador invertidos. Esta ordem pode facilitar a análise de compiladores e outros utilitários de análise, bem como remover a ambiguidade da declaração. Por exemplo, o código a seguir ilustra uma função básica em Go.

```
1  package main

3  import "fmt"

5  func add(x int) int {
       value := 42
7      return x + value
   }
9
   func main(){
11     g := add(12)
       fmt.Println(g)
13 }
```

Código A.2 – Código exemplo para declaração de função.

## A.3 TIPOS DE DADOS

Tipos de dados se referem a um sistema extensivo usado para declarar variáveis ou funções de diferentes tipos. O tipo de variável determina quanto espaço ocupa no armazenamento e como o padrão de bits armazenado é interpretado. Go suporta os seguintes tipos de dados: **boolean**, **numeric**, **string**, **integer** e **floating**.

## A.4 TIPOS ENCAPSULANTES

O Go fornece uma série de tipos internos para uso, são:

- **Arrays** são declarados estaticamente e seu tamanho precisa ser constante;
- **Slices** por sua vez são instanciados dinamicamente, ele pode ser considerada como um vetor: uma matriz que pode diminuir ou aumentar em tempo de execução;
- **Maps** são outro tipo de contêiner de dados interno, mapeando chaves e valores.

## A.5 CONTROLE DE FLUXO

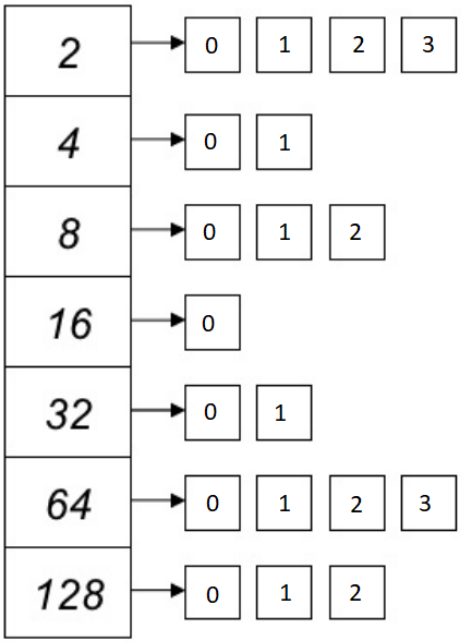
Go também fornece uma variedade de mecanismos para alterar o fluxo de controle de um programa, como: **if-then-else**, **switch** e **loops** (**for**).

## A.6 LISTA LIVRE SEGREGADA

O sistema de coleta precisa de acesso rápido à lista livre para novas alocações. Go usa uma lista livre segregada, que é um maneira de alocar objetos de forma dinâmica. A *heap* é dividida em classes de blocos de memória, cada bloco de memória pode conter um número de objetos, todos com o mesmo tamanho.

Os objetos de tamanho similar são alocados um após o outro, a posição da lista livre é indicada pelo seu índice, à medida que novos objetos precisam ser alocados a posição do índice vai aumentando. Cada nova alocação é direcionada ao primeiro bloco que tem o tamanho que satisfaça o valor do objeto que será alocado (ILIEV; KYURKCHIEV, 2018).

Figura 26 – Exemplo de lista livre segregada.




**Fonte:** o autor.

Para cada alocação em que não haja mais classes de blocos do tamanho respectivo, solicita-se mais memória do sistema operacional. A Figura 26 ilustra como é dividida a lista que, na verdade, são diversas listas onde cada uma apresenta um tamanho diferente 2, 4 ,8 e assim por diante (YOUNIS, 2012), isso pode reduzir o problema de fragmentação, porém não o elimina. Outro fator característico dessa abordagem em Go é que as listas podem ser criadas durante o tempo de execução dos programas, variando seu tamanho de acordo com a necessidade das alocações.

# APÊNDICE B – DOCKERFILE

Código B.1 – Dockerfile

```
1
FROM gcc474:latest
3
RUN apt-get update && \
5     apt-get install -y locate libgmp3-dev nano gdb wget

7 RUN mkdir -p /usr/libmago
WORKDIR /usr/libmago
9 COPY libmago/ /usr/libmago
RUN make
11
RUN wget --no-check-certificate https://dl.google.com/go/go1.9.4.linux-amd64.tar.gz
13 RUN tar -xvf go1.9.4.linux-amd64.tar.gz
RUN cp -r go /usr/local
15 ENV GOROOT=/usr/local/go
ENV GOROOT_BOOTSTRAP=/usr/libmago/go
17
ENV PATH="/usr/local/go/bin:${PATH}"
19 RUN mkdir /usr/tmp

21 CMD ["/bin/bash"]

23 ## HOW TO
##Install docker
25
#sudo apt-get install \
27 #     apt-transport-https \
#     ca-certificates \
29 #     curl \
#     software-properties-common
31

33 #curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

35 #sudo apt-key fingerprint 0EBFCD88

37
#sudo add-apt-repository \
39 #    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
#    $(lsb_release -cs) \
41 #    stable"

43 #sudo apt-get update

45 #sudo apt-get install docker-ce

47 #sudo usermod -aG docker $USER
##logout and login again
49
```

```
#gitlab image pull
#docker login registry.gitlab.com

#docker pull registry.gitlab.com/filipevarjao/go-rbmm-refcount

##creating a container using an image FOR THE FIRST TIME
#docker run --name rbmm -it registry.gitlab.com/filipevarjao/go-rbmm-refcount
##OR use image id
#docker run --name rbmm -it b97b5c7

##resuming an already created container
#docker start -i rbmm
```