



UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS GABRIEL BRITO SILVEIRA

**ANÁLISE DO COLETOR DE LIXO DA LINGUAGEM DE PROGRAMAÇÃO GO E
SEU IMPACTO NA LATÊNCIA E VAZÃO DAS APLICAÇÕES**

QUIXADÁ

2025

LUCAS GABRIEL BRITO SILVEIRA

ANÁLISE DO COLETOR DE LIXO DA LINGUAGEM DE PROGRAMAÇÃO GO E SEU
IMPACTO NA LATÊNCIA E VAZÃO DAS APLICAÇÕES

Trabalho de conclusão de curso apresentado ao
Curso de Graduação em Ciência da Computação
do Campus Quixadá da Universidade Federal
do Ceará, como requisito parcial à obtenção do
grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. João Marcelo Uchôa de
Alencar.

QUIXADÁ

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

- S589a Silveira, Lucas Gabriel Brito.
Análise do impacto do coletor de lixo da linguagem de programação Go e seu impacto na latência e vazão das aplicações / Lucas Gabriel Brito Silveira. – 2025.
53 f. : il. color.
- Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2025.
Orientação: Prof. Dr. João Marcelo Uchôa de Alencar.
1. Coleta de lixo. 2. Golang. 3. Vazão. 4. Latência. 5. Benchmarks. I. Título.

CDD 004

LUCAS GABRIEL BRITO SILVEIRA

ANÁLISE DO COLETOR DE LIXO DA LINGUAGEM DE PROGRAMAÇÃO GO E SEU
IMPACTO NA LATÊNCIA E VAZÃO DAS APLICAÇÕES

Trabalho de conclusão de curso apresentado ao
Curso de Graduação em Ciência da Computação
do Campus Quixadá da Universidade Federal
do Ceará, como requisito parcial à obtenção do
grau de bacharel em Ciência da Computação.

Aprovada em: 27/02/2025

BANCA EXAMINADORA

Prof. Dr. João Marcelo Uchôa de
Alencar (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Francisco Heron de Carvalho Junior
Universidade Federal do Ceará (UFC)

Prof. Dr. Lucas Ismaily Bezerra Freitas
Universidade Federal do Ceará (UFC)

Aos meus avós.

À minha família.

À ciência

AGRADECIMENTOS

Agradeço primeiramente aos meus avós, Maria Ivonete e João Eudes, por me incentivarem e educarem durante toda a minha vida. Os quais sempre apoiaram nos momentos difíceis e repreenderam quando necessário.

A minha mãe, Maria Marleide, por estar presente e dar tudo de si por mim e minha educação, sempre estando presente. As minhas tias e ao meu tio por todo apoio durante a graduação e fases da vida.

Expresso minha gratidão ao meu orientador, Prof. Dr. João Marcelo Uchôa de Alencar, professor admirável, o qual foi excelente na orientação deste trabalho, sendo sensível e incisivo, sempre disposto a ajudar.

Agradeço aos professores Dr. Francisco Heron de Carvalho Junior e Dr. Lucas Ismaily Bezerra Freitas por aceitarem participar da banca examinadora deste trabalho.

Por fim, agradeço aos amigos que me acompanharam na universidade (sem a presença de vocês, as filas do RU e do ônibus teriam sido extremamente longas e solitárias), e em especial ao Prof. Dr. Bruno Riccelli dos Santos Silva, que me apoia desde a época do ensino técnico.

Não se deixe prender pela necessidade de conseguir alguma coisa. Desse modo, você conseguirá tudo. (Frank Herbert, 1969)

RESUMO

Ao contrário de linguagens de programação como Java, a linguagem Go possui apenas um algoritmo de coleta de lixo implementado na distribuição oficial. Entretanto, este algoritmo pode ser ajustado através de variáveis de ambiente. Neste trabalho buscamos avaliar o comportamento do coletor de lixo da linguagem Go no âmbito de vazão e latência. O objetivo é compreender se é justificável a linguagem possuir apenas um coletor de lixo disponível. Foram utilizados dados gerados a partir da execução de *benchmarks* conceituados da linguagem Java, a qual é referência em trabalhos sobre os impactos da coleta de lixo. Tais *benchmarks* executaram com diferentes valores de GOGC, variável que permite customizar a frequência da execução do coletor, variando 20 vezes no intervalo de 50 a 1000. Os resultados mostraram que parametrizar a coleta de lixo não obteve melhorias significativas nas métricas escolhidas. Porém para aplicações com alto uso de coleta de lixo foi obtido uma redução considerável no tempo de execução. Trabalhos futuros incluem implementar diferentes coletores de lixo para o Go e realizar comparativos em diferentes cenários com o coletor atual.

Palavras-chave: coleta de lixo; Golang; vazão; latência; *benchmarks*

ABSTRACT

Unlike programming languages like Java, the Go language only has one garbage collection algorithm implemented in the official distribution. However, this algorithm can be adjusted through environment variables. In this work we seek to evaluate the behavior of the Go language garbage collector in terms of throughput and latency. The objective is to understand whether it is justifiable for the language to have only one garbage collector available. Data generated from the execution of renowned benchmarks of the Java language were used, which are a reference in works on the impacts of garbage collection. Such benchmarks are executed with different GOGC values, variable that allow customizing the frequency of collector execution, varying 20 times in the range of 50 to 1000. The results showed that parameterizing garbage collection did not obtain improvements in the chosen analyses. However, for applications with high use of garbage collection, a specific reduction in execution time was obtained. Future work includes implementing different garbage collectors for Go and comparing them in different scenarios with the current collector.

Keywords: garbage collection; Golang; throughput; latency; benchmarks

LISTA DE FIGURAS

Figura 1 – Exemplo de <i>heap</i>	17
Figura 2 – Exemplo de variável de <i>heap</i> perdida.	17
Figura 3 – Linha do tempo	20
Figura 4 – <i>Semispace</i> - Início	21
Figura 5 – <i>Semispace</i> - Cópia	22
Figura 6 – <i>Semispace</i> - Finalizado	22
Figura 7 – Contagem de referências	25
Figura 8 – Contagem de referências - Estruturas cíclicas	25
Figura 9 – <i>Mark and Sweep</i> - Etapa inicial	28
Figura 10 – <i>Mark and Sweep</i> - Marcação	29
Figura 11 – <i>Mark and Sweep</i> - Varredura	29
Figura 12 – ZGC - Ciclo	30
Figura 13 – Saída Código-fonte 3	32
Figura 14 – Exemplo de Abstração Tricolor	33
Figura 15 – Ilustração das etapas	34
Figura 16 – Fluxograma	39
Figura 17 – Quantidade de coletas de lixo realizadas por <i>benchmark</i>	45
Figura 18 – <i>parmnemonics</i> - Vazão	45
Figura 19 – <i>parmnemonics</i> - Latência	45
Figura 20 – <i>parmnemonics</i> - Tempo de execução	46
Figura 21 – <i>parmnemonics</i> - <i>Heap</i> alocado	46
Figura 22 – <i>xalan</i> - Vazão	46
Figura 23 – <i>xalan</i> - Latência	46
Figura 24 – <i>xalan</i> - Tempo de execução	47
Figura 25 – <i>xalan</i> - <i>Heap</i> alocado	47
Figura 26 – <i>fj-kmeans</i> - Vazão	47
Figura 27 – <i>fj-kmeans</i> - Latência	47
Figura 28 – <i>fj-kmeans</i> - Tempo de execução	48
Figura 29 – <i>fj-kmeans</i> - <i>Heap</i> alocado	48

LISTA DE ALGORITMOS

Algoritmo 1 – Semispace	23
Algoritmo 2 – <i>Reference Counting</i>	24
Algoritmo 3 – <i>Mark</i>	27
Algoritmo 4 – Sweep	28
Algoritmo 5 – Estrutura geral dos <i>benchmarks</i>	43

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – <i>Memory leak</i> ou variável perdida.	18
Código-fonte 2 – Referência solta.	18
Código-fonte 3 – Exemplo código Go	32

LISTA DE ABREVIATURAS E SIGLAS

CIRC	<i>Concurrent Immediate Reference Counting</i>
CSV	<i>Comma-Separated Values</i>
GC	<i>Garbage Collector</i>
JVM	<i>Java Virtual Machine</i>
SMR	<i>Safe Memory Reclamation</i>
STW	<i>Stop The World</i>
ZGC	<i>Z Garbage Collector</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	Objetivos	15
1.1.1	<i>Objetivo Geral</i>	15
1.1.2	<i>Objetivos específicos</i>	15
2	FUNDAMENTAÇÃO TEÓRICA	16
2.1	Gerenciamento de Memória	16
2.2	Coletores de Lixo	18
2.2.1	<i>Semispac</i>	21
2.2.2	<i>Reference Counting</i>	24
2.2.3	<i>Mark and Sweep</i>	26
2.2.4	<i>Z Garbage Collector (ZGC)</i>	29
2.3	Coleta de Lixo na Linguagem Go	31
3	TRABALHOS RELACIONADOS	35
3.1	<i>BestGC: An Automatic GC Selector</i>	35
3.2	<i>Assessing Contemporary Automated Memory Management in Java – Garbage First, Shenandoah, and Z Garbage Collectors Comparison</i>	36
3.3	<i>Concurrent GCs and Modern Java Workloads: A Cache Perspective</i>	37
3.4	Quadro Comparativo entre os Trabalhos Relacionados	38
4	METODOLOGIA	39
4.1	Configuração do Ambiente de Execução	39
4.2	Estudo dos <i>Benchmarks</i> Oficiais do Go	40
4.3	Reescrita dos <i>Benchmarks</i> Java Selecionados para Go	41
4.4	Coleta dos Dados de Execução	42
4.5	Análise dos Dados	42
5	EXPERIMENTOS E RESULTADOS	43
5.1	Escolhas na Reescrita dos <i>Benchmarks</i>	43
5.1.1	<i>Estrutura comum entre os benchmarks</i>	43
5.1.2	<i>parmnemonics</i>	44
5.1.3	<i>xalan</i>	44
5.1.4	<i>fj-kmeans</i>	44

5.2	Análise dos Dados	44
5.2.1	<i>parmnemonics</i>	45
5.2.2	<i>xalan</i>	46
5.2.3	<i>fj-kmeans</i>	47
5.3	Considerações Finais	48
6	CONCLUSÕES E TRABALHOS FUTUROS	49
	REFERÊNCIAS	50

1 INTRODUÇÃO

Linguagens de programação modernas incorporam o conceito de alocação dinâmica de memória, o que possibilita a criação e liberação de objetos mesmo quando a quantidade total de memória necessária para a execução não é conhecida durante a compilação do programa (Jones *et al.*, 2023). Essa característica permite que esses objetos tenham um tempo de vida maior do que a sub-rotina que os criou. Um benefício imediato é a possibilidade de utilizar técnicas de programação que empregam estruturas recursivas como árvores e *maps*. As estruturas recursivas são essenciais para o desenvolvimento de *software* atual, pois permitem a mudança em tempo de execução do tamanho de um objeto. Tais objetos são armazenados em uma área de memória conhecida como *heap* ou monte.

Em contraste com linguagens mais antigas como *COBOL*, *Basic* e *FORTRAN*, nas quais o desenvolvedor aloca a memória de forma estática, antes da compilação, linguagens como *C*, *Pascal* e *C++* permitem alocação e liberação dinâmica de memória, sob responsabilidade do desenvolvedor (Drozdek, 2000). O conceito de Coleta de Lixo (em inglês *Garbage Collection*) surge como uma forma de realizar de forma automática a liberação de memória alocada que não está em uso e nem será mais acessada.

Drozdek (2000) explica que o uso de coletores de lixo teve início por volta do ano de 1959 com a linguagem LISP. Atualmente os coletores de lixo são utilizados em uma vasta gama de linguagens de programação como Java, Go, Lua, PHP entre outras. Em trabalhos recentes como Beronić *et al.* (2022) e Tavakolisomah *et al.* (2023) os autores realizam uma comparação entre os coletores de lixo presentes na linguagem Java, com o intuito de analisar o comportamento de cada coletor tendo em vista o impacto da coleta de lixo em aplicações não triviais em ambientes com memória disponível controlada.

Os trabalhos citados no parágrafo anterior utilizam do fato do Java possuir uma vasta gama de coletores de lixo em suas versões. Beronić *et al.* (2022) analisam em quais cenários os principais coletores presentes no OpenJDK tem vantagem de desempenho sobre outros, Tavakolisomah *et al.* (2023) ressaltam a dificuldade em escolher o coletor mais apropriado, desenvolvendo assim uma metodologia para realizar de forma automática a análise e recomendação de um *Garbage Collector* (GC) com base no perfil da aplicação alvo. Ambos os trabalhos demonstram a importância e impacto na escolha do GC tem na aplicação.

Hunt e John (2011) abordam exemplos em que um ajuste fino na coleta de lixo do Java impacta diretamente o mundo real, um caso citado consiste no uso de *heap* adaptativo para

aumentar vazão, útil em sistemas de armazenamento e edição cooperativa de documentos, uma vez que sistema minimiza o uso de recursos em períodos de baixa demanda e aumenta a vazão durante uso intensivo. Patros *et al.* (2018) apontam o impacto que a coleta de lixo pode ter em ambientes *cloud* onde o uso inapropriado de recursos gera custo ao consumidor, propondo técnicas para mitigar o impacto de GC em ambientes de contêineres.

O Go utiliza de uma variação concorrente do algoritmo *Mark and Sweep*¹. Na literatura há uma escassez de trabalhos que analisam o impacto da coleta de lixo na linguagem, além disso não há informações sobre outros coletores que possam ser usados além do padrão. Desta forma, a principal motivação deste trabalho é analisar diferentes frequências na chamada do coletor de lixo da linguagem, uma vez que implementar um novo algoritmo a partir do zero é inviável no curto prazo. Estudaremos o comportamento do coletor do Go com base em *benchmarks* que analisam a coleta de lixo no Java, pois a mesma possui diferentes coletores de lixo disponíveis. O objetivo é verificar em quais cenários este ajuste no coletor padrão do Go pode trazer melhorias, possibilitando um melhor aproveitamento de recursos, almejando assim beneficiar projetistas e programadores que desenvolvem aplicações Go.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é investigar o coletor de lixo da linguagem Go analisando o comportamento do mesmo em *benchmarks* usados para avaliar os coletores do Java.

1.1.2 Objetivos específicos

- Compreender o funcionamento dos *benchmarks* de coleta de lixo já inclusos na distribuição oficial da linguagem Go.
- Codificar em Go *benchmarks* utilizados em trabalhos científicos que analisam a coleta de lixo em Java.
- Analisar o comportamento da coleta de lixo do Go diante a execução de *benchmarks* levando em consideração as métricas de latência e vazão.

¹ <https://tip.golang.org/doc/gc-guide>

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção apresenta-se o referencial teórico utilizado para compreender este trabalho. A seção está dividida em três partes: Gerenciamento de Memória, no qual será apresentado as estruturas utilizadas para fazer uso da memória principal do computador; Coletores de Lixo, na qual indicamos diferentes abordagens utilizadas em algoritmos de coleta de lixo; e por fim Coleta de Lixo na Linguagem Go, na qual exploramos seu funcionamento, vantagens, desvantagens e implementação.

2.1 Gerenciamento de Memória

Tanenbaum e Bos (2014) explicam que em sistemas operacionais modernos um processo é dividido na memória em três partes: texto, dados e pilha, sendo que neste trabalho focaremos no uso da seção de dados, especificamente no *heap*, na qual a memória alocada dinamicamente pelo programa é armazenada. O *heap* pode ser um *array* contíguo ou um conjunto de blocos não contíguos. Jones *et al.* (2023) definem um **objeto** como uma célula em uso por um programa e célula como o menor grupo de uma ou duas palavras útil suscetível de ser alocado ou liberado. Em Schildt (1997) e Drozdek (2000) os autores exploram o uso de ponteiros nas linguagens de programação, os quais os autores definem como variáveis que armazenam um endereço de memória. Neste trabalho utilizamos do termo referência para referir a um ponteiro que aponta para um objeto do *heap*.

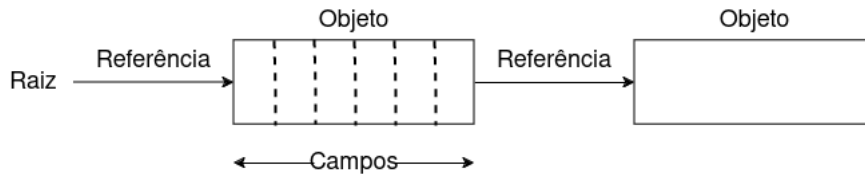
A Figura 1 exemplifica a estrutura do *heap*, na qual a raiz consiste no objeto mais externo do *heap*, e as arestas consistem em referências para os objetos. Segundo Weninger *et al.* (2018) as raízes consistem em objetos que não podem ser removidos do *heap*, por exemplo, campos estáticos e variáveis locais. na linguagem Java por exemplo, são raízes: variáveis locais, campos de classes estáticas, *threads* ativas, e referências JNI (*Java Native Interface*)¹.

Cada objeto é dividido em campos, podendo esses campos conter ou não uma referência para outro objeto. Drozdek (2000) explica a forma como os campos são organizados pode variar dependendo do coletor de lixo em questão. Um exemplo prático do uso desses campos pode ser encontrado no processo de marcação no algoritmo *Mark and Sweep*. Dependendo da implementação, pode-se utilizar um campo para verificar se o objeto já foi ou não visitado durante o processo de marcação.

¹ <https://dev.java/learn/jvm/tool/garbage-collection/java-specifics/>

Schildt (1997) aponta os riscos oriundos de linguagens de programação nas quais fica a cargo do programador alocar e liberar memória manualmente. O problema de **referência solta** ocorre quando a variável referência aponta para uma região já liberada. Já o problema da **variável de heap perdida** consiste de uma região de memória que não foi liberada, mas não é acessível por nenhuma referência ativa no programa.

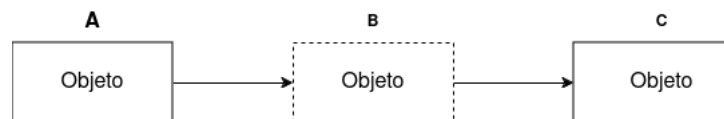
Figura 1 – Exemplo de *heap*.



Fonte: Adaptado de Jones *et al.* (2023).

A Figura 2 demonstra um cenário no qual temos um objeto A, que aponta para um objeto B e o mesmo referencia um objeto C. O problema acontece ao desalocar o objeto B, pois neste caso a referência para C é perdido e não é possível recuperar a memória alocada. Temos então uma variável de *heap* perdida.

Figura 2 – Exemplo de variável de *heap* perdida.



Fonte: Adaptado de Jones *et al.* (2023).

O Código-fonte 1 reproduz o comportamento expresso na Figura 2. Ao utilizar de uma lista encadeada com três nós, criados nas linhas 2-4, respectivamente, e em seguida deletar o nó B na linha 5, temos que não é possível acessar ou liberar o objeto C. Como a memória só poderá ser liberada ao fim da execução do programa, temos uma situação também conhecida como *memory leak*.

Já no Código-fonte 2 realizamos alocação de memória na linha 5, posteriormente utilizamos a variável criada, mas ao passar para uma função que por algum motivo libera a memória, no exemplo, a função *createDanglingPointer* libera a memória alocada, resultando em um caso de referência solta. Pois ainda podemos utilizar a variável *danglingPtr*.

Na linha 8 ao utilizar a função *print*, tentamos exibir o valor de uma variável liberada resultando em um comportamento inesperado, uma vez que pode ser exibido qualquer informação.

Código-fonte 1 – *Memory leak* ou variável perdida.

```

1     struct Node* head = createNode('A');
2     head->next = createNode('B');
3     head->next->next = createNode('C')
4     deleteNode(&(head->next));

```

Fonte: Elaborado pelo autor.

Código-fonte 2 – Referência solta.

```

1 void createDanglingPointer(int** ptr) {
2     free(*ptr);
3 }
4 int main() {
5     int* danglingPtr = (int *)malloc(sizeof(int));
6     *danglingPtr = 42;
7     createDanglingPointer(&danglingPtr);
8     printf("%d", *danglingPtr);
9 }

```

Fonte: Elaborado pelo autor.

Embora os exemplos anteriores sejam simples e a primeira vista facilmente evitáveis, os problemas oriundos de deixar a cargo do programador a alocação e liberação de memória continuam atuais. Em 2016 o Google publicou em seu *blog* de segurança sobre uma falha de segurança na função `getaddrinfo()` da *glibc* possibilitando ataques a servidores do tipo *man-in-the-middle*², falha ocasionada por um erro no tratamento manual da memória alocada.

2.2 Coletores de Lixo

O Coletor de Lixo é o componente responsável por liberar a memória alocada pelo programa que deixou de estar em uso, permitindo assim que o programa não use quantidades excessivas de memória e não sofra de problemas como *memory leaks* e referências soltas (Jones *et al.*, 2023). Em Office of the National Cyber Director (2024) e Jones *et al.* (2023), os autores afirmam que estabelecer requisitos para que programas tenham acessos seguros a memória

² <https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html>

(também conhecido como *memory safety*) são de extrema importância para garantir segurança de no desenvolvimento de *software*.

Dentre os coletores existem diferenças devido aos diversos algoritmos de alocação, pois a própria presença de coletores impacta em como o código é escrito, uma vez que o código tende a usar mais da *heap*. A coleta de lixo deve permitir alocação de memória pelo programa e reconhecer os objetos que estão e não estão em uso, ao mesmo tempo que libera a memória que não está em uso (Jones *et al.*, 2023). Coletores variam em comportamento, podendo ser *Stop The World* (STW), concorrentes, paralelos e de tempo real.

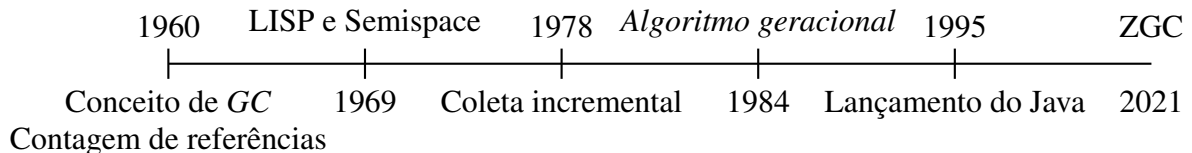
- ***Stop the World Collectors***: necessitam parar totalmente a execução da aplicação para realizar a limpeza de objetos não mais utilizados. A linguagem Java em sua versão SE 5 e SE 6 utilizava do coletor *STW* chamado *Serial*. (Wang, 2022). Tais coletores podem eventualmente melhorar o acesso por organizar os dados de forma a serem melhor encaixados na memória *cache* (Carpen-Amarie *et al.*, 2023).
- ***Concurrent Collectors***: coletores concorrentes utilizam de compartilhamento de recursos com a aplicação, buscando reduzir o tempo de pausas realizadas para a coleta de lixo. as *threads* do coletor executam em um processo e a aplicação e suas *threads* executam em um processo distinto (Beroni^ć *et al.*, 2022). As *threads* de aplicação são conhecidas como *mutators*.
- ***Parallel Collectors***: sobre coletores paralelos Beroni^ć *et al.* (2022) explica que ao interromper temporariamente a aplicação principal e utilizar de múltiplas *threads* para fazer a limpeza rapidamente os coletores paralelos conseguem aproveitar melhor o poder de processamento do sistema, reduzindo assim o tempo necessário para limpar.
- ***Real Time Collectors***: em sistemas de tempo real, o tempo dedicado a execução dos coletores pode impactar na aplicação principal, devido a limitações de tempo e necessidade de um sistema previsível tornou-se necessário adaptar coletores para este ambiente (Jones *et al.*, 2023).

Alguns coletores particionam as informações presentes no *heap* em divisões chamadas de gerações, recebendo o nome de **geracionais**, os coletores G1 e Shenandoah são coletores geracionais e não geracionais, respectivamente (Beroni^ć *et al.*, 2022). Coletores geracionais dividem os objetos presentes no *heap* com base na sua idade, a grande maioria dos coletores geracionais utiliza da hipótese da geração fraca (*weak generational hypothesis*) na qual é suposto que a maioria dos objetos tende a morrer pouco tempo depois da sua criação (Jones *et al.*, 2023).

Em outras palavras, objetos mais novos tendem a ser coletados em vez de mais antigos, e que caso um objeto viva o bastante ele acaba por envelhecer.

A Figura 3 mostra de forma simplificada um pequeno histórico no desenvolvimento de técnicas de coleta de lixo. Desde o *semispace* com a linguagem LISP (Fenichel; Yochelson, 1969) até ao coletor ZGC explorado em detalhes por Yang e Wrigstad (2022).

Figura 3 – Linha do tempo



Fonte: Adaptado de: McCarthy (1978), Jones *et al.* (2023).

Em McCarthy (1960), o autor introduziu o conceito de *Garbage Collection* ao implementar a linguagem de programação LISP, a primeira linguagem a incorporar esse mecanismo para gerenciamento automático de memória. Paralelamente Collins (1960) trabalhou na contagem de referências como método de limpar memória sem uso. Nos anos 70 houveram trabalhos como o de Cheney (1970) no qual o autor trabalhou no algoritmo de cópia de maneira não recursiva. Já no fim da década de 70, em Dijkstra *et al.* (1978) os autores trabalharam a coleta de lixo incremental, um método que permite que o coletor trabalhe em pequenos passos, reduzindo a latência do programa.

Ungar (1984) introduziu o conceito de coleta de lixo geracional, já em 1995 o lançamento da linguagem Java, com o coletor *Serial* (geracional e projetado para aplicações *single-thread*). Nos anos 2000 com a popularização de arquiteturas multicore levou ao desenvolvimento de coletores de lixo paralelos e concorrentes, como o G1 Garbage Collector, projetado para minimizar pausas em aplicações críticas.

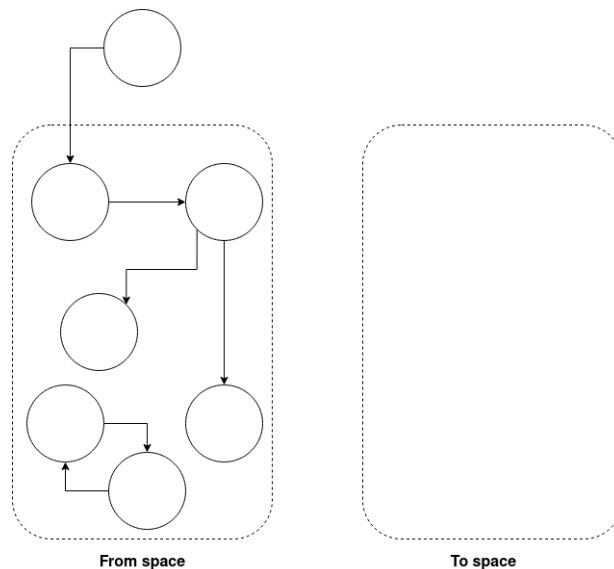
Em 2018 a RedHat lançou o *Shenandoah*, posteriormente a Oracle lançou o Z Garbage Collector (ZGC), ambos voltados para sistemas de baixa latência. Embora os coletores recentes não tratem de algoritmos novos, usando técnicas recentes conseguem bom desempenho no que se dispõem a entregar, trabalhos como o de Beronić *et al.* (2022) comparam o desempenho do G1 ao lado do ZGC e Shenandoah.

2.2.1 *Semispac*e

O algoritmo de *semispac*e consiste em dividir a memória *heap* em duas áreas distintas, denominadas *from space* e *to space*. Durante o processo de alocação, os objetos são criados na partição *from space*. Quando é necessário realizar a coleta de lixo, o algoritmo efetua uma limpeza movendo os objetos ainda em uso para a região *to space*. Esse processo de marcação envolve analisar a partir das raízes do programa e identificar os objetos acessíveis, seguindo um procedimento similar ao algoritmo *Mark and Sweep* (Seção 2.2.3). Objetos inacessíveis são então descartados, mas ao invés de varrer a área *from space*, ela é completamente substituída (Fenichel; Yochelson, 1969).

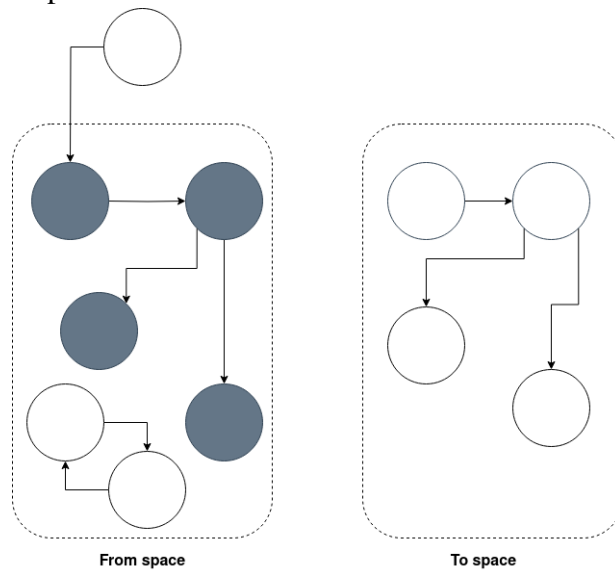
Jones *et al.* (2023) explicam que embora o algoritmo *semispac*e permita uma alocação veloz e evite problemas relacionados a fragmentação de memória, além de serem mais simples de implementar que o coletor *Mark and Sweep* o principal ponto negativo é o alto consumo de memória virtual.

Figura 4 – *Semispac*e - Início



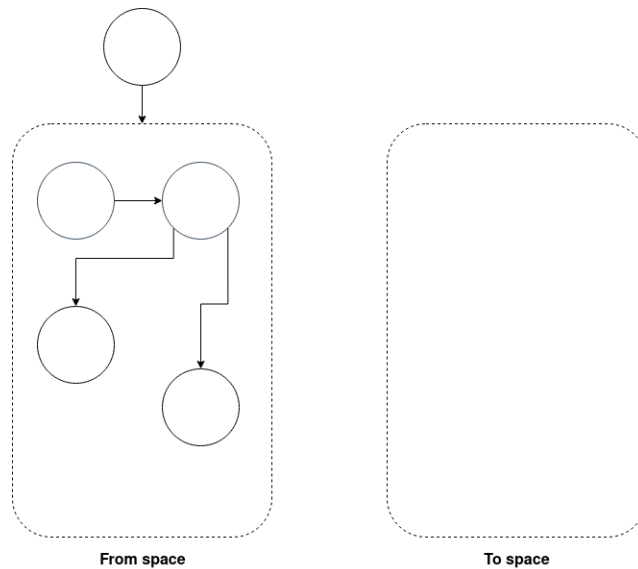
Fonte: Elaborado pelo autor.

A Figura 4 ilustra o momento inicial do algoritmo, no qual o *heap* encontra-se está particionado nas regiões *to space* e *from space*. A Figura 5 demonstra o processo de marcação, onde a raiz aponta para um objeto, e este por sua vez aponta para outro objeto, por fim a Figura 6 avança na execução do Algoritmo 1 a fim de mostrar o percurso deste processo, no qual o algoritmo percorre e copia os objetos da partição *from space* para o espaço *to space*.

Figura 5 – *Semispace* - Cópia

Fonte: Elaborado pelo autor.

No Algoritmo 1 a função *collect* chama a função *flip* responsável por inverter os espaços *to space* e *from space* e após isso inicializar a *worklist* com os objetos diretamente acessíveis pelas raízes. No laço presente na linha 4 o algoritmo percorre cada campo nas raízes e verifica se o objeto é ativo. Já no laço da linha 7 o algoritmo percorre todos os elementos da *worklist* buscando por elementos alcançáveis por meio da função *scan*.

Figura 6 – *Semispace* - Finalizado

Fonte: Elaborado pelo autor.

Algoritmo 1: Semispace

```

1 Function collect():
2   flip();
3   initialise(worklist);
4   for each fld in Roots do
5     | process(fld);
6   end
7   while not isEmpty(worklist) do
8     | ref ← remove(worklist);
9     | scan(ref);
10  end
11 Function flip():
12  fromspace, tospace ← tospace, fromspace;
13  top ← tospace + extent;
14  free ← tospace;
15 Function process(fld):
16  fromRef ← *fld;
17  if fromRef ≠ null then
18    | *fld ← forward(fromRef);
19  end
20 Function scan(ref):
21  for each fld in Pointers(ref) do
22    | process(fld);
23  end
24 Function forward(fromRef):
25  toRef ← forwardingAddress(fromRef);
26  if toRef = null then
27    | toRef ← copy(fromRef);
28  end
29  return toRef;
30 Function copy(fromRef):
31  toRef ← free;
32  free ← free + size(fromRef);
33  move(fromRef, toRef);
34  forwardingAddress(fromRef) ← toRef;
35  add(worklist, toRef);
36  return toRef;

```

Fonte: Adaptado de Jones *et al.* (2023).

2.2.2 Reference Counting

A técnica de contagem de referências (*reference counting*) diferente dos algoritmos *Mark and Sweep* e *semispace*, nos quais o grafo de objetos é percorrido através das raízes e os objetos não visitados são considerados lixo, consiste em assumir que um objeto está vivo se o número de referências a esse objeto for maior que zero (Jones *et al.*, 2023), A Figura 7 exemplifica este funcionamento. Linguagens atuais como Python³ e Perl⁴ utilizam de contagens de referência e em versões modernas o C++ permite o uso de ponteiros inteligentes⁵, os quais utilizam de contagem de referência para automatizar a liberação de memória, como forma de evitar problemas relacionados a alocação manual de memória.

Algoritmo 2: Reference Counting

```

1 Function new () :
2   | ref ← allocate();
3   | if ref = null then
4     |   error ← Out of memory;
5   | end
6   | rc(ref) ← 0 ;
7   | return ref ;
8 Function write (src, i, ref) :
9   | addReferenceref;
10  | deleteReferencesrc[i];
11  | src[i] ← ref;
12 Function addReference (ref) :
13  | if ref ≠ null then
14    |   rc(ref) ← rc(ref) + 1 ;
15  | end
16 Function deleteReference (ref) :
17  | if ref ≠ null then
18    |   rc(ref) ← rc(ref) - 1 ;
19    |   if rc(ref) = 0 then
20      |     forall fld ∈ Pointers(ref) do
21        |       end
22        |       deleteReference*fld ;
23      |   end
24      |   free(ref)
25  | end

```

Fonte: Adaptado de Jones *et al.* (2023).

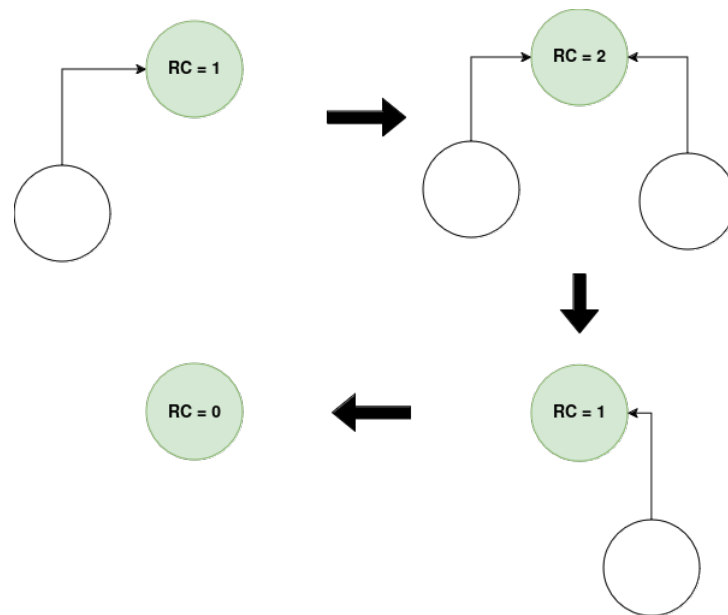
³ <https://docs.python.org/3/c-api/refcounting.html>

⁴ <https://perldoc.perl.org/perlref>

⁵ https://en.cppreference.com/book/intro/smart_pointers

O Algoritmo 2 exemplifica uma codificação básica, onde as referências são incrementadas ou decrementadas conforme a criação e destruição de objetos. A função *Write* aumenta a contagem de referência do novo objeto e depois diminui a contagem do objeto antigo. As funções *addReference* e *deleteReference* incrementam e decrementam, respectivamente, as contagens de referência do objeto. Uma vez que a contagem de referência é zero, o objeto pode ser liberado e as contagens de referência de todos os seus filhos diminuídas.

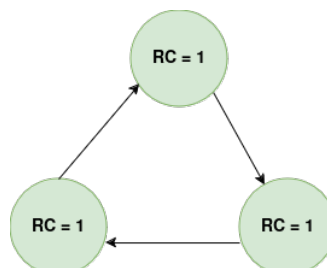
Figura 7 – Contagem de referências



Fonte: Elaborado pelo autor.

A contagem de referências segundo Jones *et al.* (2023) e Jung *et al.* (2024) tem benefícios como: potencial para conseguir reciclar uma alta quantidade de memória, bom desempenho e facilidade de implementação. Porém uma desvantagem considerável é a incapacidade de lidar com estruturas cíclicas, como a representada pela figura 8 uma vez que mesmo os objetos sendo inacessíveis sua contagem nunca será zerada, fazendo assim com que nunca seja removida do *heap*.

Figura 8 – Contagem de referências - Estruturas cíclicas



Fonte: Elaborado pelo autor.

Jones *et al.* (2023) aponta a técnica *trial deletion* como uma maneira de lidar com estruturas cíclicas, uma vez que ela consiste em não observar o grafo de maneira completa, e sim apenas a parte onde liberar uma referência pode causar uma estrutura cíclica. O coletor trabalha sobre subgrafos, formados a partir de objetos identificados como possíveis estruturas cíclicas. De maneira similar ao algoritmo de Dijkstra *et al.* (1978) os objetos visitados são pintados de cinza e tem pontos diminuídos na contagem de referências, objetos que estão vivos são pintados de preto e demais objetos de branco. Todos os objetos do subgrafo que terminam brancos são considerados lixo e são liberados.

Trabalhos recentes como o de Jung *et al.* (2024) e Anderson *et al.* (2021) exploram a contagem de referências, enquanto Anderson *et al.* (2021) trabalhou em uma maneira de juntar contagem de referências e *Safe Memory Reclamation* (SMR) atingindo bom desempenho ao aplicar no C++, no trabalho de Jung *et al.* (2024) os autores desenvolveram a técnica *Concurrent Immediate Reference Counting* (CIRC) que permite uma maneira concorrente, rápida e segura de trabalhar com estruturas encadeadas longas.

2.2.3 *Mark and Sweep*

O algoritmo *Mark and Sweep* apresentado em Jones *et al.* (2023) é um coletor não geracional, constituído de duas principais fases, *Mark* e *Sweep*, marcar e varrer, respectivamente. Na fase de marcação é necessário percorrer o *heap* a partir de cada raiz. O pseudo-código demonstrado no Algoritmo 3 exemplifica uma implementação serial da fase de marcação, enquanto o Algoritmo 4 trata de uma implementação da fase de varredura, também de forma serial.

Na linha 13 a *worklist* é inicializada como vazia. Esta lista será usada para rastrear os objetos que precisam ser marcados. Após essa etapa no laço da linha 14, o algoritmo itera sobre cada referência em *Roots* (raízes), sendo que esta estrutura *Roots* representa os pontos de partida para a marcação, inicializada no começo do processo de coleta, quando todos os *mutators* são pausados e *field* representa os campos presentes nas raízes. Para cada campo a referência é obtida, no condicional da linha 16 se a referência não for nula e ainda não estiver marcada, o algoritmo marca a referência e a adiciona a *worklist*.

Algoritmo 3: Mark

```

1 Function mark () :
2   while not isEmpty(worklist) do
3     ref ← remove(worklist);
4     for each field in Pointers(ref) do
5       child ← *fld;
6       if child ≠ null and not isMarked(child) then
7         setMarked(child);
8         add(worklist, child);
9       end
10    end
11  end
12 Function markFromRoots (Roots) :
13   Input :Roots
14   Output :Marking from roots
15   worklist ← empty;
16   for each field in Roots do
17     ref ← *field;
18     if ref ≠ null and not isMarked(ref) then
19       setMarked(ref);
20       add(worklist, ref);
21       mark();
22     end
23   end

```

Fonte: Adaptado de Jones *et al.* (2023).

Após trabalhar sobre as raízes, o algoritmo chama a função *mark*, que processa todos os elementos na *worklist* continuando a marcar objetos acessíveis transitivamente a partir das raízes. Na função *mark()* na *worklist* estão os objetos acessíveis diretamente a partir das raízes, enquanto a *worklist* não estiver vazia, o algoritmo continua a processar os objetos na lista, removendo o próximo objeto lista e passa a iterar sobre todas as referências contidas no objeto. Para cada referência, se ela não for nula e ainda não estiver marcada, o algoritmo marca a referência e a adiciona na *worklist*. Todo o processo descrito garante que o objeto referenciado será processado posteriormente para que suas próprias referências também sejam marcadas, garantindo a eficácia da coleta.

No Algoritmo 4, na linha 2 a variável *scan* é inicializada com o valor de *start*, por sua vez *start* e *end* representam o endereço da posição inicial e final das regiões do *heap* a serem analisadas. Continua-se a varrer a memória enquanto *scan* for menor que *end*. Dentro do *loop* na linha 3, verifica-se *scan* está marcado. Se o objeto está marcado, o algoritmo desmarca o objeto, para que a próxima execução do *mark* ocorra. Se o objeto não está marcado, a memória é liberada, a variável *scan* é atualizada para apontar para o próximo objeto na memória.

Algoritmo 4: Sweep

```

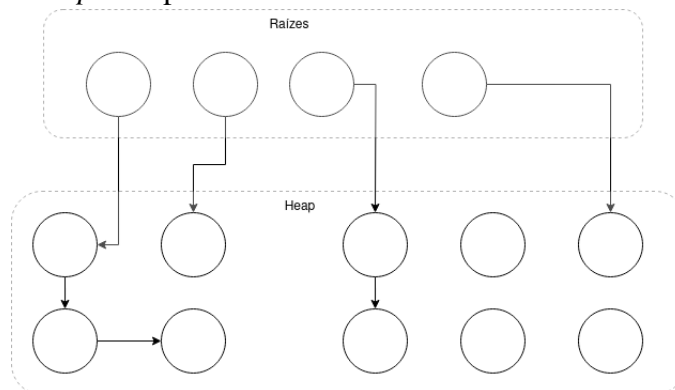
1 Function sweep (start, end) :
   Input : start, end
   Output : Sweeping from start to end
2 scan ← start;
3 while scan < end do
4   if isMarked(scan) then
5     | unsetMarked(scan);
6   end
7   else
8     | free(scan);
9   end
10  scan ← nextObject(scan);
11 end

```

Fonte: Adaptado de Jones *et al.* (2023).

A Figura 9 exemplifica o *heap* antes da execução da fase de marcação, já a Figura 10 exhibe o *heap* após o processo de marcação, onde cada objeto que é alcançável a partir das raízes é marcado. Durante a etapa de varredura (*sweep*), é necessário percorrer novamente o *heap* e liberar os objetos que não foram marcados na fase anterior. A Figura 11 ilustra o estado final do *heap* após a conclusão da fase de *sweep*, onde os objetos não utilizados foram removidos.

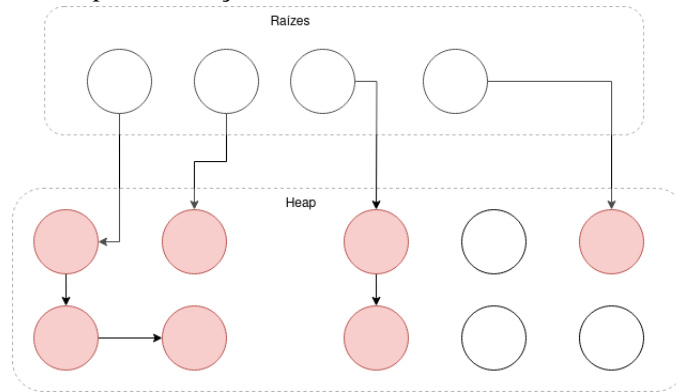
Figura 9 – *Mark and Sweep* - Etapa inicial



Fonte: Elaborado pelo autor.

O processo de marcação pode ser feito utilizando de um *bitmap* ou de uma *flag* no *header* do objeto, caso o objeto não tenha referências salvas em seus campos não o adicionamos na *worklist*, porém ele ainda pode ser marcado. Na implementação serial podemos lidar com a *worklist* como uma pilha, o que ajuda na localidade das informações na cache. Embora Carpen-Amarie *et al.* (2023) demonstrem que em aplicações Java reais o impacto de localidade no *cache* não é um problema de desempenho considerável, o *Mark and Sweep* é ruim no quesito localidade.

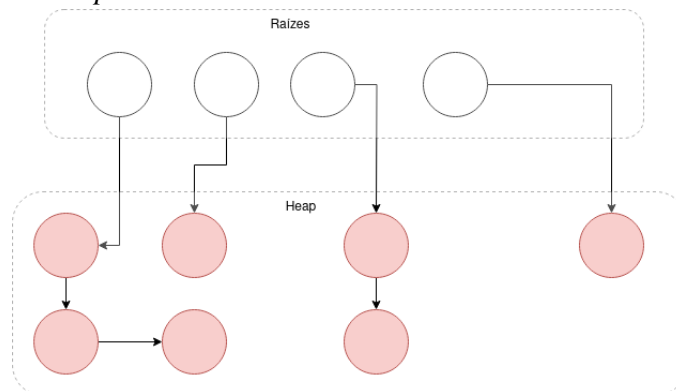
Figura 10 – *Mark and Sweep* - Marcação



Fonte: Elaborado pelo autor.

A fase de marcação encerra quando todos os objetos alcançáveis a partir das raízes são marcados, o algoritmo é finito pois eventualmente a *worklist* ficará vazia, qualquer objeto não marcado é considerado lixo. A fase de varredura geralmente é linear, passando por cada nó não marcado e o marcando, ao passo que em nós marcados troca o *status* do marcador para que o próximo ciclo de coleta ocorra sem impedimentos.

Figura 11 – *Mark and Sweep* - Varredura



Fonte: Elaborado pelo autor.

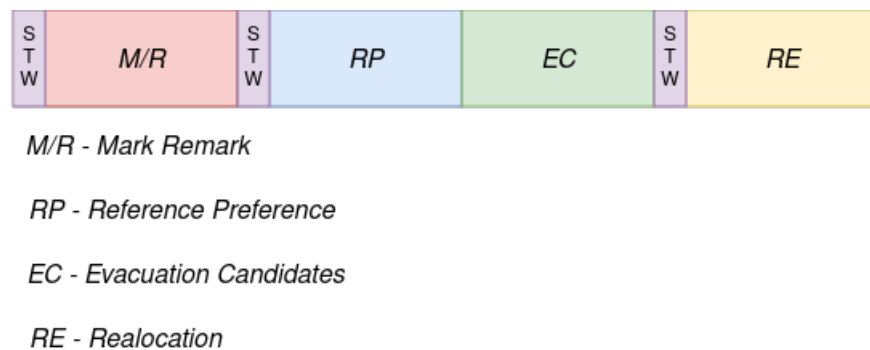
2.2.4 Z Garbage Collector (ZGC)

Z Garbage Collector (ZGC) é um coletor de lixo atual, lançado em sua versão experimental para o JDK 11. Este coletor realiza a coleta de maneira não geracional, usando de paralelismo e concorrência, conseguindo lidar com *heap* de até 16TB com tempo de pausa fixo⁶. O ZGC expande algoritmos clássicos como o *Semispace* e *Mark and Sweep*, aliado a técnicas modernas como paralelismo e concorrência para conseguir ganhos de desempenho.

⁶ <https://wiki.openjdk.org/display/zgc>

Yang e Wrigstad (2022) aprofundam-se no funcionamento do ZGC, ressaltando as técnicas e escolhas presentes no coletor, como por exemplo a técnica de ponteiros coloridos, que consiste em visualizar os *bits* de um ponteiro (suporta apenas arquiteturas 64-*bits*) em duas partes, representando a cor do ponteiro e o endereço real de memória. Um ponteiro pode ter uma dentre diversas cores, sendo que uma cor *boa* indica que o ponteiro é válido e pode ser desreferenciado de forma segura, por sua vez uma cor considerada *ruim* indica que o ponteiro pode não ser válido. No ZGC o que é uma cor *boa* ou *ruim* varia dependendo do momento em que a coleta de lixo encontra-se.

Figura 12 – ZGC - Ciclo



Fonte: Adaptado de Yang e Wrigstad (2022).

Outro componente importante no ZGC é o uso de *self-healing barriers*, seu uso permite ter certeza que as *threads* de aplicação apenas trabalharam com ponteiros válidos, permitindo consertar casos de referência pendente oriundos do ciclo de execução. A Figura 12 exemplifica o ciclo de funcionamento do ZGC, composto por quatro fases concorrentes intercaladas por pausas *Stop The World* (STW) (Yang; Wrigstad, 2022). Em resumo temos que:

- **STW-1:** As *threads* definem uma cor considerada *boa*, sendo que a cor definida como *boa* pode mudar no decorrer da execução. Raízes são marcadas com uma cor *boa*, são checadas e atualizadas, após isso são adicionadas a pilha de marcação.
- **Mark Remark:** Nessa fase, os objetos vivos são marcados, utilizando da pilha de marcação definida na etapa anterior para identificar objetos ainda vivos.
- **STW-2:** Consiste em fazer novas checagens e atualizações caso necessário, validando o estado do *heap*.
- **Reference Preference:** Nessa fase as referências são trabalhadas a fim de verificar se todos os objetos acessíveis foram marcados. usando de *self-healing barriers* para corrigir eventuais ponteiros ruins.

- **Evacuation candidates:** O coletor identifica quais objetos podem ser movidos para outras localizações na memória, essa fase leva em consideração o resultado da fase de marcação, uma vez que somente objetos vivos podem ser realocados.
- **STW-3:** Ocorre de maneira similar a STW2
- **Reallocation:** Por fim, os objetos que foram movidos na fase anterior são alocados em novas páginas de memória e tem suas referências atualizadas.

Além do trabalho de Yang e Wrigstad (2022), Beroníć *et al.* (2022) e Tavakolisomeh *et al.* (2023) constataram a boa performe do ZGC em *heaps* consideráveis, além de sua baixa latência.

2.3 Coleta de Lixo na Linguagem Go

Como citado nas seções anteriores, existem diferentes abordagens nos coletores de lixo. A linguagem Go usa do *Mark and Sweep* de forma concorrente. Esta seção busca explicar como o Go lida com a alocação de memória e detalhes de implementação do seu coletor de lixo.

De acordo com (Go Programming Language, 2024) o compilador do Go é o responsável por escolher se uma variável irá para o *heap* ou a pilha, determinando também as raízes. Exemplos de raízes são variáveis locais e globais. Por padrão variáveis que não são referências não costumam serem tratadas pelo coletor de lixo, ficando a cargo do compilador determinar quando tais variáveis serão livres. Os valores que não podem ter a memória alocada na pilha, vão para o *heap*.

Quando uma variável é alocada de forma dinâmica, por exemplo, *arrays* e *slices* menores que 10MB são alocados na pilha, enquanto maiores vão para o *heap*. O Código-fonte 3 e a Figura 13 demonstram esse cenário. No Código-fonte 3 definimos variáveis que têm tamanho próximos ao limite de 10MB. Na compilação do programa com a *flag* adequada, temos o sumário da localização de cada variável na Figura 13.

Código-fonte 3 – Exemplo código Go

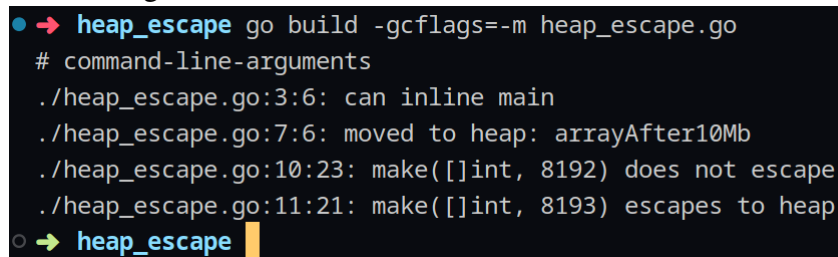
```

1 func main() {
2     var arrayBefore10Mb [1310720]int
3     arrayBefore10Mb[0] = 1
4     var arrayAfter10Mb [1310721]int
5     arrayAfter10Mb[0] = 1
6     sliceBefore64 := make([]int, 8192)
7     sliceOver64 := make([]int, 8193)
8     sliceOver64[0] = sliceBefore64[0]
9 }

```

Fonte: Elaborado pelo autor.

Figura 13 – Saída Código-fonte 3



```

● → heap_escape go build -gcflags=-m heap_escape.go
# command-line-arguments
./heap_escape.go:3:6: can inline main
./heap_escape.go:7:6: moved to heap: arrayAfter10Mb
./heap_escape.go:10:23: make([]int, 8192) does not escape
./heap_escape.go:11:21: make([]int, 8193) escapes to heap
○ → heap_escape

```

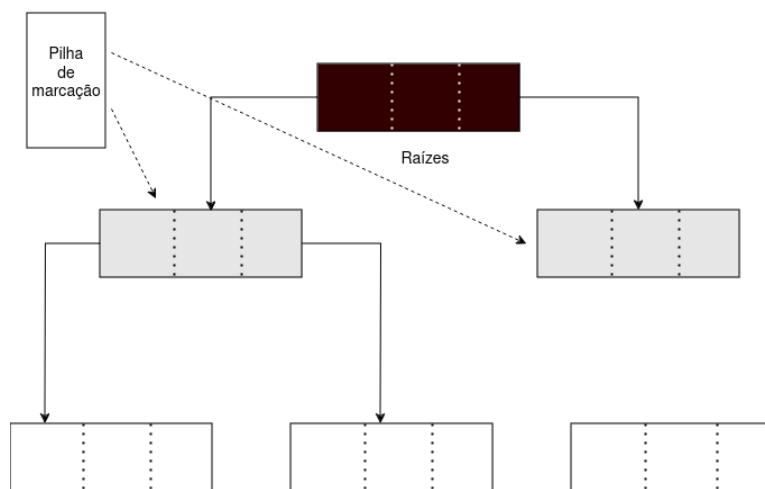
Fonte: Elaborado pelo autor.

No algoritmo *Mark and Sweep* explicado na seção anterior consideramos uma execução serial, porém isso significa aumentar o tempo de pausas STW, pois analisamos o *heap* como algo estático no momento da coleta. Ao considerar concorrência, consideramos que entre a Figura 10 e a Figura 11 um *mutator* (*thread* de aplicação) adiciona mais elementos ao *heap*, não sendo possível determinar se os objetos adicionados já foram analisados. No Go com o intuito de permitir coleta concorrente é utilizado uma variação do Algoritmo *Mark and Sweep* chamada de Abstração Tricolor.

A Abstração Tricolor, proposta por Dijkstra *et al.* (1978), consiste em particionar os objetos do grafo em preto (vivo), branco (provavelmente morto) e cinza. Objetos brancos são fortes candidatos a serem limpos, uma vez que nenhum objeto encontrado a partir das raízes, objetos pretos foram analisados e são encontrados a partir das raízes, objetos cinza estão em processo de análise.

Quando um nó é encontrado pela primeira vez durante a fase de marcação ele é colorido de cinza, ao ser percorrido e possuir filhos é pintado de preto. Ao considerar um *mutator* como um objeto, ao atribuir cores podemos representar se as raízes foram ou não escaneadas, um *mutator* cinza significa que ainda não foi terminado de ser escaneado, enquanto um preto significa que suas raízes já foram escaneadas. O progresso na marcação é feito separando os objetos pretos dos brancos até que todos os objetos pesquisáveis através das raízes se tornem pretos. Objetos cinza são analisados novamente, sendo adicionados a uma pilha de marcação.

Figura 14 – Exemplo de Abstração Tricolor

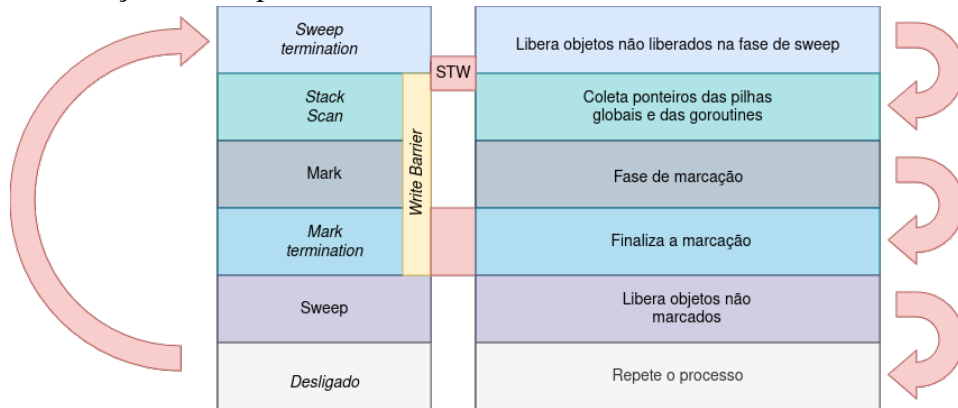


Fonte: Adaptado Jones *et al.* (2023).

A Figura 14 exibe um exemplo de marcação utilizando abstração tricolor, no qual a pilha de marcação é responsável por guardar objetos em processamento. Objetos pintados de preto já foram finalizados, objetos cinza estão em processamento, e objetos brancos ainda serão visitados, caso não sejam visitados serão livres.

O algoritmo de coleta do Go pode então ser dividido em 5 passos: *Sweep Termination*, *Stack Scan*, *Mark*, *Mark Termination* e *Sweep*, nas quais de forma cíclica ao terminar a última iteração, prepara para a próxima execução. A fase de *Sweep Termination*, libera objetos que não foram liberados na fase de varredura anterior. Isso ocorre somente quando o ciclo de coleta de lixo é interrompido abruptamente. No início da fase de *Mark*, é habilitada uma *Write Barrier* (mecanismo utilizado para garantir a correteza das referências). A Figura 15 ilustra o processo descrito acima, onde as colunas vermelhas representam os períodos de pausas STW e a coluna amarela o período onde a *Write Barrier* está ativa.

Figura 15 – Ilustração das etapas



Fonte: Adaptado de Hudson (2015).

Conforme a Figura 15, durante a marcação das raízes todas as pilhas são escaneadas, e quaisquer ponteiros para o *heap* são marcados de cinza, indicando que precisam ser processados. Cada objeto cinza é escaneado e marcado de preto, sinalizando que foram completamente processados, enquanto todos os ponteiros encontrados nesses objetos são marcados de cinza. O processo continua até que não existam mais objetos cinzas ou tarefas de marcação de raízes pendentes. A fase de *Mark Termination* ocorre quando todos os objetos cinzas foram processados, concluindo assim a fase de marcação. Na fase de *Sweep*, a *write barrier* é desativada. Novos objetos alocados são pintados de branco, indicando que ainda não foram processados. Objetos que não foram marcados durante a fase de marcação são liberados, recuperando a memória.

3 TRABALHOS RELACIONADOS

Os trabalhos relacionados citados a seguir trabalham de forma prática o impacto de diferentes GC em uma vasta gama de cenários, levando em consideração os conceitos apresentados no Seção 2 e o ambiente de execução. O primeiro trabalho aborda a problemática da escolha de um coletor adequado para uma aplicação levando em conta seu perfil. O segundo trabalho busca comparar os coletores mais relevantes presentes no OpenJDK e analisar em quais cenários um coletor leva a ganhos de desempenho em uma aplicação. Por fim o terceiro trabalho trata de uma análise do impacto na memória *cache* causado pelo uso de coletores de lixo.

Os trabalhos compartilham elementos comuns, como a utilização de suítes de *benchmark* famosas no Java e os coletores alvos de estudo. As suítes comum entre os trabalhos são a suíte DaCapo, explorada em detalhes por Blackburn *et al.* (2006), e a suíte Renaissance explorada por Prokopec *et al.* (2019). Embora os coletores estudados variem entre os trabalhos, são estudados alguns coletores presentes no OpenJDK sendo eles: ZGC profundamente analisado e estudado por Yang e Wrigstad (2022), G1 explorado por Detlefs *et al.* (2004), Shenandoah estudado por Flood *et al.* (2016) e Parallel¹.

3.1 *BestGC: An Automatic GC Selector*

Em Tavakolisomeh *et al.* (2023) os autores apresentam o desenvolvimento da ferramenta *BestGC* que tem como objetivo analisar o uso de recursos de uma aplicação Java e recomendar o GC que entrega os melhores resultados. Para a realização do estudo foi necessário elencar os coletores de lixo alvos da análise, pois existem diversos coletores de lixo feitos pela comunidade que não se encontram no OpenJDK. Entre as diferentes implementações disponíveis no OpenJDK, foram escolhidos os coletores ZGC, G1, Shenandoah e Parallel. Os autores explicam que a escolha do coletor de lixo ideal para determinada aplicação não é uma tarefa trivial, necessitando de tempo, testes sistemáticos e análise de dados. O *BestGC* surge como uma maneira de automatizar tal processo, pois assim como pode trazer melhorias, escolher um coletor pode trazer um sobrecarga maior para a aplicação, em especial aplicações críticas, nas quais cada milissegundo é de suma importância.

As métricas do estudo são: tempo de pausa, uso de memória e vazão da aplicação, aliado a execução com diferentes quantidades de memória disponíveis. A escolha das métricas

¹ <https://docs.oracle.com/en/java/javase/11/gctuning/parallel-collector1.html>

justifica-se devido a visão dos autores na qual tais métricas refletem de forma útil para o usuário o impacto da coleta de lixo, pois é possível visualizar quanto tempo uma aplicação é interrompida pelo coletor e quanto a quantidade de memória alocada impacta na execução. A quantidade de memória livre variou sendo elas 256MB, 512MB, 1024MB, 2048MB, 4096MB e 8192MB. As aplicações testadas foram escolhidas de duas suítes famosas de *benchmark* para Java, sendo elas DaCapo e Renaissance.

Os autores conseguiram desenvolver o *BestGC* de forma que o coletor sugerido trás em média 36.75% de ganho de desempenho em comparação com o coletor padrão do Java. Considerando outros coletores disponíveis na comunidade, o *BestGC* consegue sugerir o mesmo coletor de lixo em 51.24% dos casos e a melhor categoria de coletor em 85.95% dos casos. O *G1* e *Parallel* em casos gerais se comportam melhor *Shenandoah* e *ZGC* relativo ao tempo de execução, principalmente em *heaps* maiores, e também que o *ZGC* ganha em tempos de tempo de pausa, sendo seguido pelo *Shenandoah*. Um trabalho futuro será melhorar a precisão do *BestGC* e cogitam usar de aprendizado de máquina a fim de obter melhores resultados.

O trabalho de Tavakolisomeh *et al.* (2023) assemelha-se com nossa proposta ao analisar o impacto da coleta de lixo e em sua metodologia. A diferença está em termos de linguagem avaliada como alvo do estudo. Uma vez que Tavakolisomeh *et al.* (2023) utilizam da linguagem Java e desenvolvem uma aplicação para avaliar a execução e sugerir o melhor coletor de lixo de forma automatizada com base no perfil da aplicação alvo.

3.2 *Assessing Contemporary Automated Memory Management in Java – Garbage First, Shenandoah, and Z Garbage Collectors Comparison*

No trabalho de Beronić *et al.* (2022) os autores ressaltam a importância do gerenciamento de memória de forma automática, evitando problemas como *memory leaks* e referência solta que ocorrem ao deixar a cargo do desenvolvedor a liberação da memória alocada. Os autores afirmam que o conhecimento do funcionamento dos coletores de lixo é fundamental para permitir melhoria no desempenho de aplicações. Devido ao trabalho explorar a coleta na linguagem Java os autores explicam a arquitetura do Java Hotspot². Esta parte central da *Java Virtual Machine* (JVM) atua no gerenciamento da *heap*, no coletor de lixo e no compilador.

O trabalho compara três algoritmos de coleta de lixo presentes no OpenJDK: *Garbage First* (*G1*), *Shenandoah* e *ZGC*. Os autores elucidam o ciclo de execução e o *layout* da *heap*

² <https://www.oracle.com/java/technologies/javase/javase-core-technologies-apix.html>

para cada um dos três coletores, destacando as diferentes abordagens utilizadas. O estudo tem como proposta analisar o comportamento dos coletores de lixo citados acima a fim de visualizar em quais aspectos cada coletor pode trazer vantagens para as aplicações. Para a análise foram utilizados *benchmarks* das suítes DaCapo e Renaissance.

Os autores enfatizam que aplicações atuais, principalmente as que lidam com muitos dados, usam a *heap* de forma intensiva. Ressaltando assim a importância de algoritmos de coleta de lixo eficazes, entre os algoritmos analisados, o coletor *GI* tem melhor desempenho em aplicações que utilizam *heaps* menores, em contrapartida o *Shenandoah* lida melhor com *heaps* maiores. Sobre o *ZGC* devido a seu pequeno tempo de pausa, trabalha bem em aplicações com uma quantidade bem maior de dados.

Embora o trabalho de Beronić *et al.* (2022) convirja com nossa proposta por analisar diferenças no desempenho das aplicações com diferentes abordagens de coleta de lixo, eles divergem significativamente em termos de linguagem avaliada e algoritmos avaliados. Uma vez que Beronić *et al.* (2022) utilizam a linguagem Java e seus coletores de lixo presentes no OpenJDK.

3.3 *Concurrent GCs and Modern Java Workloads: A Cache Perspective*

A análise desenvolvida em Carpen-Amarie *et al.* (2023) aborda a importância da coleta de lixo, porém tem como foco os custos envolvidos no processo. O trabalho busca analisar o impacto da coleta de lixo concorrente na memória *cache*. A ideia inicial é que a coleta de lixo polui a *cache* ao executar de forma concorrente a fim de diminuir as pausas para a execução do coletor, pois a aplicação precisa gerenciar *threads* e recursos, impactando de forma negativa o desempenho da aplicação. Os autores utilizaram de uma metodologia que permitisse quantificar o impacto do coletor de lixo na *cache*, mais especificamente no último nível (*Last Level Cache - LLC*).

Os autores analisaram a correlação entre *LLC misses* e a atividade do coletor de lixo, utilizando como coletores alvos deste estudo o *Shenandoah* e *ZGC*, por se tratarem de coletores recentes, concorrentes e presentes no OpenJDK. Os *benchmarks* escolhidos foram selecionados a partir da suíte Renaissance, entre os desafios do estudo os autores destacam a falta de possibilidade de remover totalmente o coletor de lixo das aplicações e a dificuldade em medir os impactos diretos na memória *cache*.

A conclusão dos pesquisadores com base na análise dos dados foi que embora

coletores de lixo concorrentes acabem por impactar em uma grande quantidade de erros no acesso a *cache*, afetando aplicações em ambiente controlado, aplicações reais de maneira geral não tem perdas significativas de desempenho. A razão seria o tamanho da *LLC* não sendo limitado o bastante para aplicações de tamanho similar as cargas de trabalho utilizadas.

O estudo de Carpen-Amarie *et al.* (2023) embora aprofunde-se no impacto da coleta de lixo na memória *cache*, aproxima-se da nossa proposta ao estudar o impacto da coleta no desempenho das aplicações. Divergindo no uso da linguagem Java e na pesquisa focada no impacto da coleta de lixo na memória *cache*.

3.4 Quadro Comparativo entre os Trabalhos Relacionados

O Quadro 1 mostra uma síntese acerca dos trabalhos relacionados apresentado nas subseções acima.

Quadro 1 – Comparativo

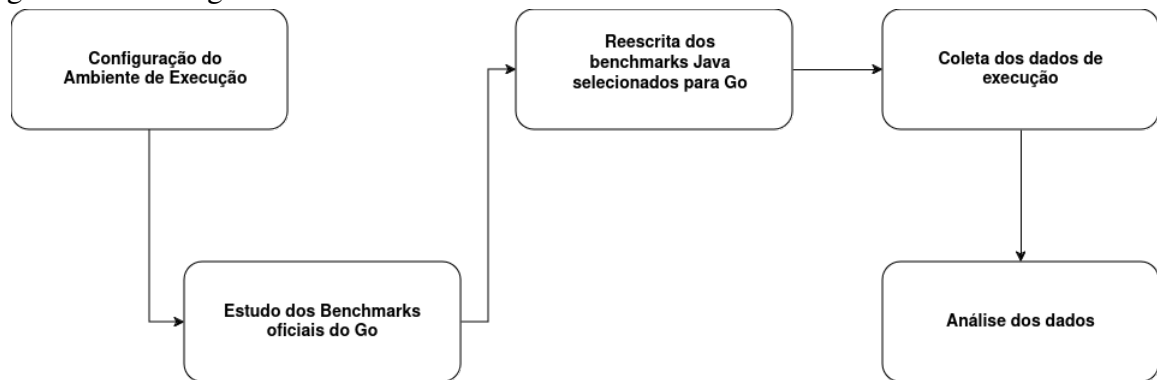
Trabalhos Relacionados	Benchmark	Linguagem Avaliada	Algoritmos Analisados	Divergências	Convergências
Tavakolisomeh <i>et al.</i> (2023)	DaCapo , Renaissance	Java	G1, Parallel, Shenandoah, ZGC	Desenvolve uma solução que automatiza a escolha do GC	Impacto do coletor de lixo e metodologia
Beronić <i>et al.</i> (2022)	DaCapo, Renaissance	Java	G1, Shenandoah, ZGC	Compara coletores da linguagem Java	Estudo do impacto do coletor de lixo e metodologia
Carpen-Amarie <i>et al.</i> (2023)	Renaissance	Java	Shenandoah, ZGC	Pesquisa do impacto da coleta de lixo na cache e uso da linguagem Java	Metodologia ao analisar o impacto da coleta na cache
Este trabalho	DaCapo, Renaissance	Go	Coletor de lixo do Go	Explora a coleta de lixo na linguagem Go	Estudo do impacto do coletor de lixo

Fonte: Elaborado pelo autor.

4 METODOLOGIA

A metodologia utilizada neste trabalho foi composta por cinco etapas. Primeiro, a configuração de um ambiente de execução controlado. Segundo, estudo dos *benchmarks* oficiais do Go. Terceiro, reescrita dos *benchmarks* Java selecionados para Go. Quarto, coleta dos dados de execução, na qual houve parametrização da execução dos *benchmarks* da etapa anterior utilizando diferentes configurações da variável de ambiente *GOGC*. Esta variável permite controlar a frequência da coleta de lixo nas aplicações. Por fim, na quinta e última etapa, de forma similar aos trabalhos de Beronić *et al.* (2022) e Tavakolisomeh *et al.* (2023), analisamos o impacto do coletor na vazão e latência, junto com o tempo total de execução.

Figura 16 – Fluxograma



Fonte: Elaborado pelo autor.

4.1 Configuração do Ambiente de Execução

Para realização dos testes o ambiente de execução consistiu em uma máquina virtual de 64 *bits* executando Ubuntu Server 24.04, utilizando quatro núcleos e 12 GB RAM. A máquina *host* tem um processador Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz com 8 *threads* e 24 GB de RAM com o sistema operacional Ubuntu Desktop 22.04. A versão escolhida do Go foi a 1.21.5 por se tratar de uma versão recente, porém que não está mais em desenvolvimento, possibilitando que a execução das aplicações disponha dos recursos e otimizações mais recentes da linguagem. A escolha de uma versão com ciclo de desenvolvimento encerrado ocorreu para evitar eventuais atualizações que possam ter algum impacto na execução das aplicações.

4.2 Estudo dos *Benchmarks* Oficiais do Go

A linguagem Go possui dois *benchmarks* em código aberto específicos para testar o funcionamento do coletor de lixo, disponibilizados no GitHub da linguagem, sendo um *benchmark* para o teste de latência ¹ e outro para testar a quantidade de coletas feitas ². Para o desenvolvimento deste trabalho, os *benchmarks* foram estudados com o intuito de entender a codificação da coleta de dados, assim possibilitando a reescrita dos *benchmarks* Java listados pelos trabalhos de Tavakolisomah *et al.* (2023) e Beronić *et al.* (2022).

A linguagem também possui no pacote *testing* com a possibilidade de escrever e executar *benchmarks* de forma simples e eficiente, entre os dados que podem ser obtidos destacam-se os que referem-se a estatísticas presentes na estrutura *MemStats*. Esta estrutura permite acessar informações sobre o uso de memória em tempo de execução, sendo alguns deles:

- **Mallocs:** representa o total de objetos que estão alocados no *heap*.
- **Frees:** representa o total de objetos que foram libertos no *heap*.
- **HeapAlloc:** quantidade de *bytes* alocados em objetos do *heap*.
- **HeapSys:** quantidade de memória em *bytes* obtidas do sistema operacional.
- **HeapReleased:** quantidade de memória em *bytes* devolvidas ao sistema operacional.
- **NextGC:** A quantidade de *heap* que fará a coleta de lixo ser disparada no próximo ciclo.
- **LastGC:** tempo em nanosegundos que desde que a última coleta de lixo encerrou.
- **PauseTotalNs:** tempo total em nanosegundos de pausas *STW* desde que o programa iniciou.
- **NumGC:** quantidade de ciclos de coleta de lixo concluídos.
- **NumForcedGC:** número de ciclos de coleta de lixo ativados pelo desenvolvedor de forma forçada.
- **GCCPUFraction:** Tempo de *cpu* usado pela coleta de lixo desde o início do programa.

Podemos afirmar que o ambiente de desenvolvimento oficial da linguagem Go já apresenta os mecanismos necessários para a coleta de informações sobre o comportamento da memória durante a execução. Portanto, para nossa proposta, podemos tirar proveito desse arcabouço.

¹ https://github.com/golang/benchmarks/tree/master/gc_latency

² <https://github.com/golang/benchmarks/blob/master/garbage>

4.3 Reescrita dos *Benchmarks* Java Selecionados para Go

Embora os *benchmarks* mencionados anteriormente sejam mantidos oficialmente pela linguagem Go, trabalhos como Beronić *et al.* (2022) e Tavakolisomeh *et al.* (2023) que possuem foco na linguagem Java, utilizam de suítes de *benchmark* diferentes para obter a carga de trabalho utilizados nos testes. Como exemplos, podemos citar as suítes *DaCapo* e *Renaissance*. Nosso trabalho fará uso de uma codificação própria em Go dos algoritmos em comum entre os trabalhos.

– **DaCapo:**

- *Xalan*: Converte documentos XML, aplicando folhas de estilo XSLT para produzir saídas formatadas.
- A escolha do Xalan se justifica por ser uma carga de trabalho customizável, além de ser usado no trabalho de (Tavakolisomeh *et al.*, 2023).

– **Renaissance:**

- *fj-kmeans*: O algoritmo de *k-means* utilizando *Fork/Join* em Java paraleliza o processo de agrupamento de dados em *clusters*. Inicialmente, os pontos de dados são distribuídos em subtarefas, cada uma responsável por calcular a distância dos pontos aos centróides e atribuir os pontos ao *cluster* mais próximo. Cada subtarefa é tratada como uma instância de uma classe que estende *RecursiveTask*. O *ForkJoinPool* gerencia a execução paralela dessas subtarefas.
- *parmnemonics*: A técnica de *pair mnemonics* utilizando *streams* do JDK para facilitar a memorização de pares de informações, como palavras ou números. Com *streams*, você pode processar coleções de dados de forma declarativa e paralela. Cada par é transformado em uma associação mnemônica significativa através de operações com mapas.

Trabalhos como o de Abhinav *et al.* (2020) e Togashi e Klyuev (2014) analisaram a performance do Go e Java. Togashi e Klyuev (2014) trabalharam utilizando de *goroutines* e *WaitGroup* na escrita do código Go. Em seus testes conseguiram desempenho melhor no Go em relação ao Java em tempo de execução. Em Abhinav *et al.* (2020) os autores escreveram dois *benchmarks* utilizando de canais e *goroutines*, os autores concluem que o Java performa melhor em aplicações com muita carga de trabalho e pouca criação de *threads*, já o Go em aplicações que demandam criação de *threads* de maneira contínua independente da quantidade de trabalho. Em ambos os trabalhos foi ressaltado o custo mais baixo na criação de *threads* no Go.

Para a reescrita dos *benchmarks* escolhidos foi utilizado das técnicas utilizadas no trabalho de Togashi e Klyuev (2014) e Abhinav *et al.* (2020), utilizando de *goroutines*, canais e *WaitGroup* para adaptação do código Java para Go.

4.4 Coleta dos Dados de Execução

Para a obtenção dos dados, foi utilizado os resultados dos *benchmarks* produzidos na etapa anterior, uma vez que seu uso permitiu coletar os dados brutos que foram explorados na etapa de análise. Os *benchmarks* foram executados 100 vezes para cada valor da variável GOGC utilizada, a qual variou de 50 a 1000 com razão 50. Os dados produzidos foram armazenados em um arquivo *Comma-Separated Values* (CSV).

4.5 Análise dos Dados

Neste trabalho a comparação entre os resultados entre os coletores ocorreu levando em consideração vazão e latência. Hunt e John (2011) definem **vazão** como a medida da quantidade de trabalho que pode ser realizado por unidade de tempo, e **latência** como a medida do tempo decorrido entre quando uma aplicação recebe um estímulo para realizar algum trabalho e quando esse trabalho é concluído. Neste trabalho trabalhamos com vazão como a quantidade de memória liberada ao sistema operacional por unidade de tempo, sendo a mesma calculada por *HeapReleased* em razão do tempo total de execução em nanossegundos, já a latência foi calculado como o total de tempo pausado em nanossegundos dividido pela quantidade de coletas realizadas.

A escolha de latência como parâmetro ocorreu devido ao fato de que embora coletores paralelos e coletores concorrentes consigam diminuir a quantidade de vezes que a aplicação do usuário para totalmente em determinadas aplicações o tempo de resposta baixo é um requisito crucial, já a escolha de vazão ocorreu pois a devolução de memória ao sistema evita problemas de falta de memória. Em trabalhos como o de Zhao *et al.* (2022) os autores exploram a relação entre latência e vazão ao produzirem um coletor focado em alta vazão e baixa latência, os autores afirmam a importância no balanceamento entre essas características, principalmente em aplicações sensíveis ao tempo de resposta.

5 EXPERIMENTOS E RESULTADOS

5.1 Escolhas na Reescrita dos *Benchmarks*

A reescrita dos algoritmos selecionados das suítes *DaCapo* e *Reinassance* teve como principal desafio tentar manter o código Go o mais semelhante possível com o código em sua versão Java. Os *benchmarks* escritos seguem um fluxo comum, aliado a estrutura de *benchmark* do pacote de testes nativo da linguagem Go, no qual é definido um laço o qual itera uma quantidade definida pelo usuário. Dentro é feito duas chamadas do coletor de lixo da aplicação, após isso o *benchmark* executa e seus dados de execução são salvos para serem escritos no arquivo de resultados. Vale ressaltar que durante os testes foi buscado deixar o tempo de execução de cada algoritmo similar. Os algoritmos gerados neste trabalho estão disponíveis no repositório do projeto ¹.

5.1.1 Estrutura comum entre os *benchmarks*

O Algoritmo 5 exemplifica de forma geral como os *benchmarks* foram estruturados. Na linha um e dois, criamos o CSV responsável por armazenar os dados de execução, já nas linhas quatro e cinco, forçamos a execução da coleta de lixo duas vezes para garantir resultados precisos ². No restante do algoritmo, temos a execução do *benchmark* escolhido e a coleta do tempo gasto na sua execução e o salvamento dos dados coletados.

Algoritmo 5: Estrutura geral dos *benchmarks*

```

1 createFile();
2 executions_data;
3 for i := 0; i < b.N; i ++ do
4     runtime.GC();
5     runtime.GC();
6     start_time = time.Now();
7     benchmark.Run();
8     end_time = time.Since(start_time);
9     executions_data ← ReadMemStats();
10    append(executionData, end_time);
11 writeExecutionData();
```

Fonte: Elaborado pelo autor.

¹ github.com/lucasgabrielbritosilveira/benchmarks

5.1.2 *parmnemonics*

A versão implementada neste trabalho tentou aproximar-se o máximo possível da versão em Java, replicando as funções e divisão do fluxo de execução de forma similar a original. A função *Run* do *benchmark parmnemonics* consiste em realizar a codificação de uma palavra para o *mnemonic* correspondente, no qual é criada uma *goroutine* responsável por realizar o *encode* de forma paralela. Para sincronismo foi utilizado de *Waitgroups*.

5.1.3 *xalan*

A adaptação do *xalan* consistiu em, de forma similar ao original da suíte *DaCapo*, ter como carga de trabalho a transformação de XML em HTML usando um template definido. A codificação do *benchmark* consistiu em utilizar da mesma entrada da versão Java; iterar sobre os arquivos XML e, por fim, passa-los sobre o arquivo de estilo. A função finaliza quando todos os arquivos são trabalhados.

O principal desafio consistiu no fato da biblioteca *xalan* utilizada no Java não possuir contraparte no Go. A versão desenvolvida neste trabalho utilizou da biblioteca de conversão *go-xslt*² como substituta.

5.1.4 *fj-kmeans*

O *benchmark fj-kmeans* por sua vez utiliza da técnica de *Fork Join* para dividir a execução do Kmeans em partes menores usando a abordagem de divisão e conquista. Para adaptar o mecanismo de Fork/Join foram utilizado de *goroutines* junto com *WaitGroup*. O algoritmo consiste em gerar dados aleatórios e buscar por centroides, no teste são gerados 100.000 pontos e 5 *clusters* no total.

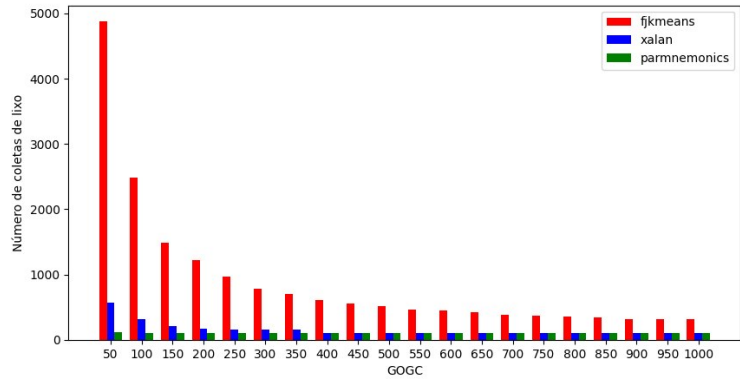
5.2 Análise dos Dados

Ao analisar os dados, foi utilizado da biblioteca *matplotlib* para geração dos gráficos com base no CSV no qual foi removido o menor e pior valor de cada métrica antes da geração dos gráficos. A Figura 17 exibe um comparativo entre a quantidade de ciclos de coleta de lixo finalizados nos *benchmarks*. Podemos observar que o *fj-kmeans* demonstra um número de coletas consideravelmente maior em relação aos demais testes.

² <https://github.com/wamuir/go-xslt>

Também é importante notar que a quantidade de coletas acompanha do valor da variável GOGC da maneira prevista, ou seja, quanto maior o valor da variável menor a interferência do coletor de lixo.

Figura 17 – Quantidade de coletas de lixo realizadas por *benchmark*

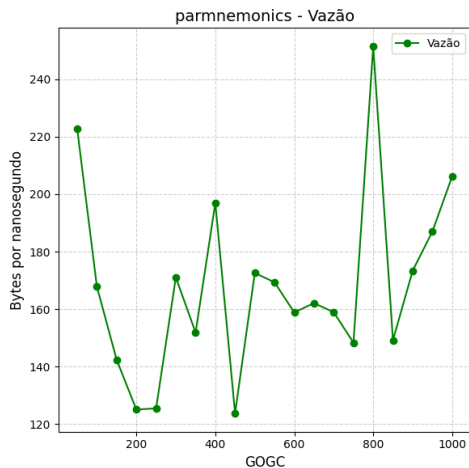


Fonte: Elaborado pelo autor.

5.2.1 *parmnemonics*

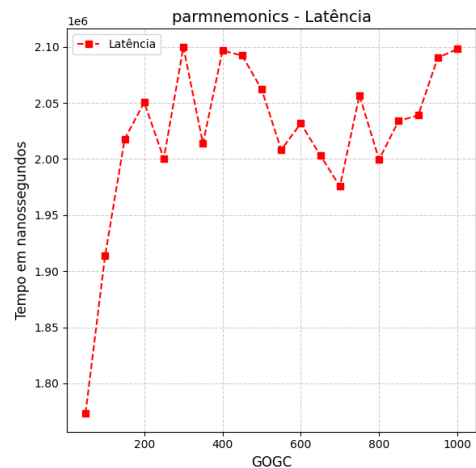
A Figura 18 demonstra o comportamento da vazão no *benchmark parmnemonics*. Visualizamos que a vazão apresenta picos e declives em relação a memória alocada demonstrada na Figura 21. Por sua vez temos que na latência, exibida na Figura 19 é possível visualizar que embora o *heap* cresça a latência permanece em um intervalo regular. A Figura 20 exibe o tempo total de execução, nela é possível visualizar que tempo de execução da aplicação não apresenta comportamento previsível. Podemos concluir que apesar de alguma regularidade, o *parmnemonics* não apresenta um comportamento definido pela variação da variável GOGC.

Figura 18 – *parmnemonics* - Vazão



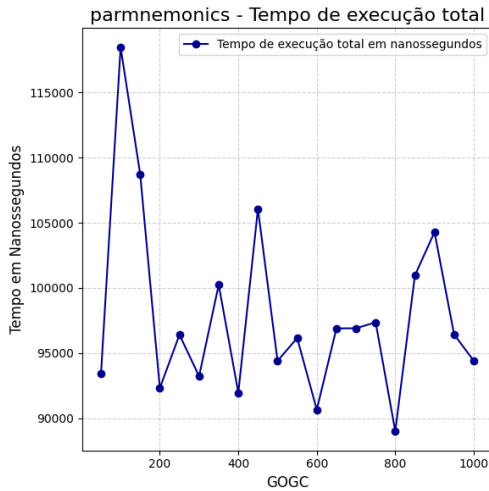
Fonte: Elaborado pelo autor.

Figura 19 – *parmnemonics* - Latência



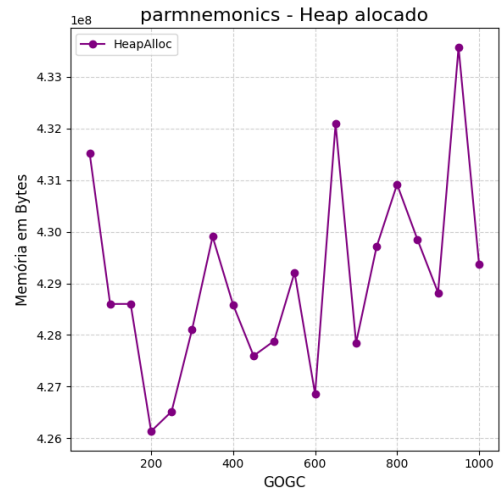
Fonte: Elaborado pelo autor.

Figura 20 – *parmnemonics* - Tempo de execução



Fonte: Elaborado pelo autor.

Figura 21 – *parmnemonics* - Heap alocado

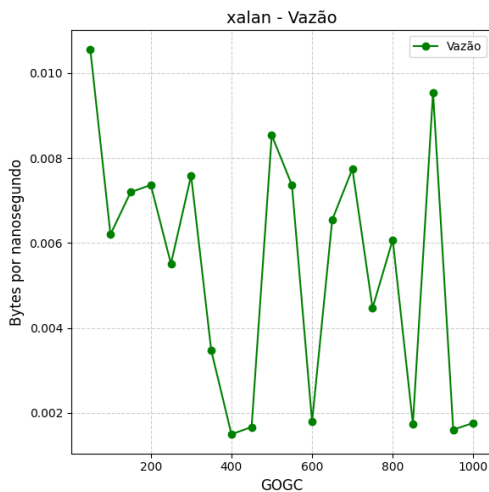


Fonte: Elaborado pelo autor.

5.2.2 *xalan*

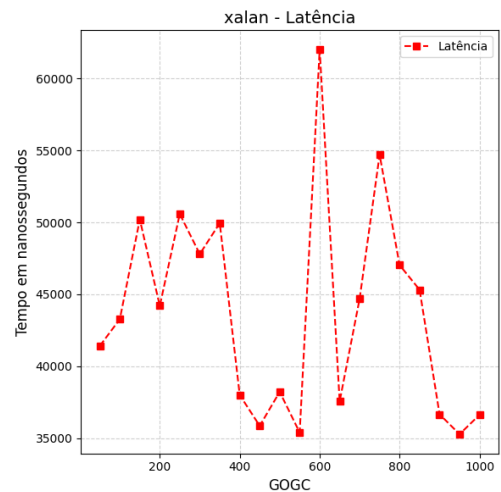
A vazão do *xalan* exibida na Figura 22 possui picos e varia de forma irregular com a variável GOGC. Em relação a latência exibida na Figura 23 conseguimos ver redução na latência para valores muito altos de GOGC, as custas de um aumento na memória alocada demonstrada na Figura 25. Já a Figura 24 exhibe o tempo total de execução. De forma geral, assim como o *parmnemonics*, o *xalan* não apresenta comportamento previsível pela variação da variável GOGC. O que é esperado visto que pela Figura 17 essas aplicações não realizam uma quantidade significativa de coleta.

Figura 22 – *xalan* - Vazão



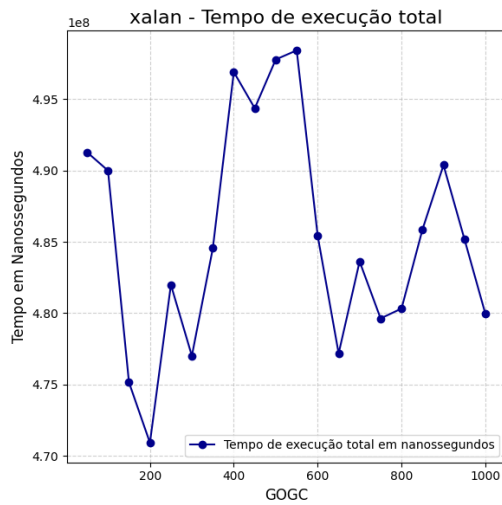
Fonte: Elaborado pelo autor.

Figura 23 – *xalan* - Latência



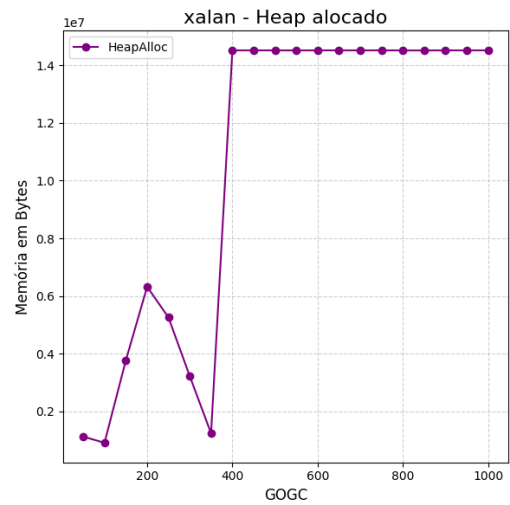
Fonte: Elaborado pelo autor.

Figura 24 – *xalan* - Tempo de execução



Fonte: Elaborado pelo autor.

Figura 25 – *xalan* - Heap alocado

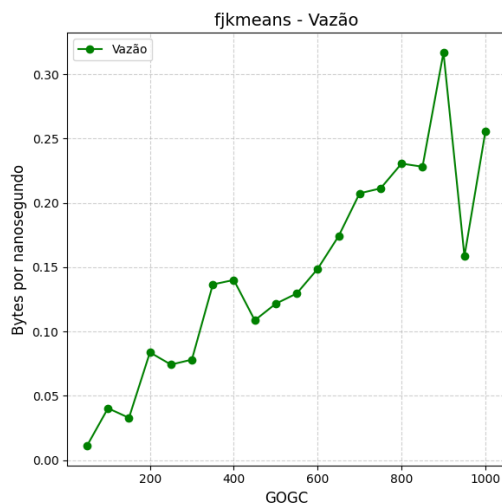


Fonte: Elaborado pelo autor.

5.2.3 *fj-kmeans*

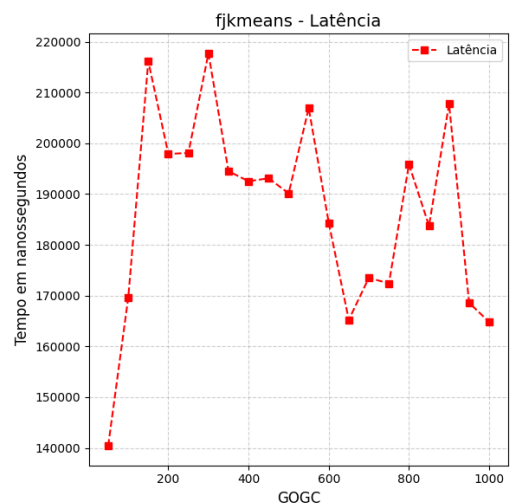
A Figura 26 demonstra o comportamento da vazão conforme a frequência da coleta de lixo diminui. Visualizamos que a vazão tende a aumentar, uma vez que a quantidade de memória alocada cresce, como demonstrado na Figura 29. Por sua vez temos que a latência, exibida na Figura 27 tende a diminuir.

Figura 26 – *fj-kmeans* - Vazão



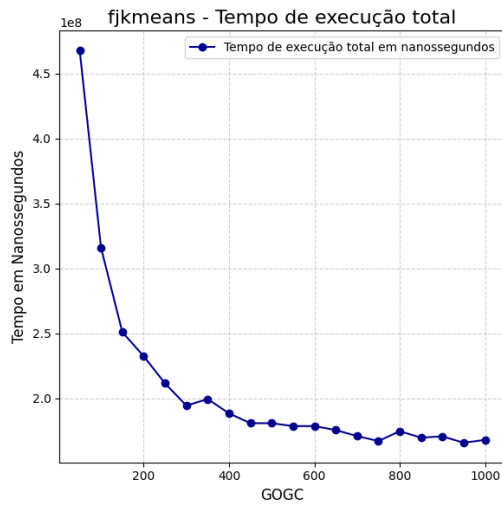
Fonte: Elaborado pelo autor.

Figura 27 – *fj-kmeans* - Latência

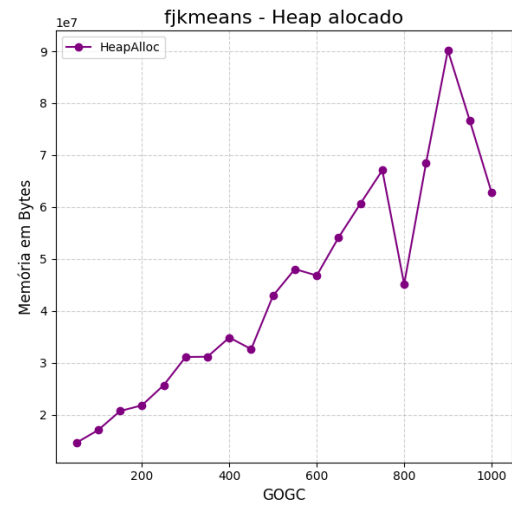


Fonte: Elaborado pelo autor.

A Figura 28 exibe o tempo total da aplicação, nela é possível visualizar a diminuição do tempo de execução da aplicação, assim caso o aumento de *heap* seja menos importante que a quantidade de tempo de execução, pode ser viável reduzir a quantidade de coleta de lixo realizada.

Figura 28 – *ff-kmeans* - Tempo de execução

Fonte: Elaborado pelo autor.

Figura 29 – *ff-kmeans* - Heap alocado

Fonte: Elaborado pelo autor.

Por ser a aplicação com maior quantidade de coletas realizadas durante a execução, podemos observar uma tendência clara de comportamento de acordo com a variável GOGC.

5.3 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos pelos experimentos com os *benchmarks: parmnemonics, xalan e ffkmeans*. Observamos o impacto da coleta de lixo do Go em aplicações que realizam quantidade elevada de coleta de lixo, em especial vazão e latência, além do impacto no tempo de execução. Os experimentos permitiram compreender e explorar o funcionamento da coleta de lixo em aplicações reais, bem como comportamentos que podem permitir um ajuste fino, caso desejado pelo usuário.

Os resultados mostraram que de maneira geral o uso da variável GOGC permite ajustes reais no *ff-kmeans*, no qual ocorreu tendência a diminuição da latência a medida que a frequência do coletor diminuiu, quanto a vazão foi visualizado estabilidade ou aumento da quantidade de memória liberada por coleta. Por sua vez, os *benchmarks xalan e parmnemonics* o comportamento foi bastante irregular.

Por fim, inferiu-se que em aplicações com baixo uso de coleta de lixo parametrizar o GOGC não trouxe ganhos. Já ao personalizar o coletor em aplicações com muitas coletas podem ser obtidos ganhos em tempo de execução. Com isso temos um coletor equilibrado que pode justificar o fato do Go não ter outros coletores, uma vez permite um nível de customização aceitável para quem deseja um ajuste fino em tempos de execução.

6 CONCLUSÕES E TRABALHOS FUTUROS

A coleta de lixo, presente em vasta gama de linguagens de programação é estudada nos dias atuais, visando permitir o maior aproveitamento de recursos minimizando o custo para a aplicação. Com isso o objetivo geral deste trabalho é de investigar o coletor de lixo da linguagem Go analisando o comportamento do mesmo em *benchmarks* usados para avaliar os coletores do Java. Dessa forma foram produzidos *benchmarks* em Go com base em equivalentes de suítes renomadas da linguagem Java, a qual é referência em coleta de lixo.

Dentre as principais dificuldades encontradas, pode-se destacar o porte dos *benchmarks* escolhidos, em especial o *xalan*, pois depende de biblioteca externa, junto com a diferença entre as linguagens Java e Go.

Sobre os resultados obtidos, o *benchmark fj-kmeans* se mostrou mais sensível a frequência da coleta de lixo. Durante os experimentos foi percebido que nos três *benchmarks* testados houve redução no tempo de execução total ao reduzir a frequência da coleta de lixo, sendo o *fj-kmeans* o mais afetado. Entretanto, o comportamento das métricas nos outros *benchmarks* foi bastante irregular. Desta forma o trabalho mostrou que em aplicações com baixo uso do coletor de lixo, a variação do GOGC não apresentou impactos significativos em vazão e latência,

Como trabalhos futuros, uma primeira proposta seria expandir a avaliação para utilizar mais aplicações na avaliação, representando um espectro maior na variação da quantidade de coleta de lixo realizada. Em seguida, a implementação de diferentes coletores de lixo para o Go e permitiria a realização de comparativos em diferentes cenários com o coletor atual com apoio de *benchmarks* mais amplos especializados em estressar a memória.

REFERÊNCIAS

- ABHINAV, P. Y.; BHAT, A.; JOSEPH, C. T.; CHANDRASEKARAN, K. Concurrency analysis of go and java. In: **2020 5th international conference on computing, communication and security (ICCCS)**. [S. l.: s. n.], 2020. p. 1–6.
- ANDERSON, D.; BLELLOCH, G. E.; WEI, Y. Concurrent deferred reference counting with constant-time overhead. In: **Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation**. [S. l.: s. n.], 2021. p. 526–541.
- BERONIĆ, D.; NOVOSEL, N.; MIHALJEVIĆ, B.; RADOVAN, A. Assessing contemporary automated memory management in java garbage first, shenandoah, and z garbage collectors comparison. In: IEEE. **2022 45th Jubilee international convention on information, communication and electronic technology (MIPRO)**. [S. l.], 2022. p. 1495–1500.
- BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHANG, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z. *et al.* The dacapo benchmarks: Java benchmarking development and analysis. In: **Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications**. [S. l.: s. n.], 2006. p. 169–190.
- CARPEN-AMARIE, M.; VAVOULIOTIS, G.; TOVLETOGLOU, K.; GROT, B.; MUELLER, R. Concurrent gcs and modern java workloads: a cache perspective. In: **Proceedings of the 2023 ACM SIGPLAN international symposium on memory management**. [S. l.: s. n.], 2023. p. 71–84.
- CHENEY, C. J. A nonrecursive list compacting algorithm. **Communications of the ACM**, ACM New York, NY, USA, v. 13, n. 11, p. 677–678, 1970.
- COLLINS, G. E. A method for overlapping and erasure of lists. **Communications of the ACM**, ACM New York, NY, USA, v. 3, n. 12, p. 655–657, 1960.
- DETFLEFS, D.; FLOOD, C.; HELLER, S.; PRINTEZIS, T. Garbage-first garbage collection. In: **Proceedings of the 4th international symposium on memory management**. [S. l.: s. n.], 2004. p. 37–48.
- DIJKSTRA, E. W.; LAMPORT, L.; MARTIN, A. J.; SCHOLTEN, C. S.; STEFFENS, E. F. On-the-fly garbage collection: nn exercise in cooperation. **Communications of the ACM**, ACM New York, NY, USA, v. 21, n. 11, p. 966–975, 1978.
- DROZDEK, A. **Data structures and algorithms in C++**. [S. l.]: Brooks/Cole Publishing Co., 2000.
- FENICHEL, R. R.; YOCHELSON, J. C. A lisp garbage-collector for virtual-memory computer systems. **Communications of the ACM**, ACM New York, NY, USA, v. 12, n. 11, p. 611–612, 1969.
- FLOOD, C. H.; KENNKE, R.; DINN, A.; HALEY, A.; WESTRELIN, R. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In: **Proceedings of the 13th international conference on principles and practices of programming on the Java platform: virtual machines, languages, and tools**. [S. l.: s. n.], 2016. p. 1–9.

Go Programming Language. **Go GC: A guide to garbage collection in go.** 2024. Disponível em: <https://tip.golang.org/doc/gc-guide>. Acesso em: 18 Ago. 2024.

HUDSON, R. **Go GC: Latency problem solved.** 2015. GopherCon Denver. Google confidential and proprietary. Disponível em: <https://go.dev/src/runtime/mgc.go>. Acesso em: 30 Jun. 2024.

HUNT, C.; JOHN, B. **Java performance.** [S. l.]: Prentice Hall Press, 2011.

JONES, R.; HOSKING, A.; MOSS, E. **The garbage collection handbook: the art of automatic memory management.** [S. l.]: CRC Press, 2023.

JUNG, J.; KIM, J.; PARKINSON, M. J.; KANG, J. Concurrent immediate reference counting. **Proceedings of the ACM on programming languages**, ACM New York, NY, USA, v. 8, n. PLDI, p. 151–174, 2024.

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

MCCARTHY, J. History of lisp. In: **History of programming languages.** [S. l.: s. n.], 1978. p. 173–185.

Office of the National Cyber Director. **Final ONCD technical report.** [S. l.], 2024. Disponível em: <https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>. Acesso em: 30 dez. 2024.

PATROS, P.; KENT, K. B.; DAWSON, M. Mitigating garbage collection interference on containerized clouds. In: IEEE. **2018 IEEE 12th international conference on self-adaptive and self-organizing systems (SASO).** [S. l.], 2018. p. 168–173.

PROKOPEC, A.; ROSÀ, A.; LEOPOLDSEDER, D.; DUBOSCQ, G.; TMA, P.; STUDENER, M.; BULEJ, L.; ZHENG, Y.; VILLAZÓN, A.; SIMON, D. *et al.* Renaissance: Benchmarking suite for parallel applications on the jvm. In: **Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.** [S. l.: s. n.], 2019. p. 31–47.

SCHILD, H. **C completo e total.** [S. l.]: Makron, 1997.

TANENBAUM, A. S.; BOS, H. **Sistemas operacionais modernos.** 4th. ed. [S. l.]: Pearson, 2014. ISBN 978-8543005676.

TAVAKOLISOMEH, S.; BRUNO, R.; FERREIRA, P. Bestgc: an automatic gc selector. **IEEE Access**, IEEE, 2023.

TOGASHI, N.; KLYUEV, V. Concurrency in go and java: performance analysis. In: **2014 4th IEEE international conference on information science and technology.** [S. l.: s. n.], 2014. p. 213–216.

UNGAR, D. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. **ACM Sigplan notices**, ACM New York, NY, USA, v. 19, n. 5, p. 157–167, 1984.

WANG, S. Java garbage collectors. **International journal of open information technologies**, . . . , v. 10, n. 6, p. 57–61, 2022.

WENINGER, M.; GANDER, E.; MÖSSENBÖCK, H. Utilizing object reference graphs and garbage collection roots to detect memory leaks in offline memory monitoring. In: **Proceedings of the 15th international conference on managed languages & runtimes**. [S. l.: s. n.], 2018. p. 1–13.

YANG, A. M.; WRIGSTAD, T. Deep dive into zgc: a modern garbage collector in openjdk. **ACM transactions on programming languages and systems (TOPLAS)**, ACM New York, NY, v. 44, n. 4, p. 1–34, 2022.

ZHAO, W.; BLACKBURN, S. M.; MCKINLEY, K. S. Low-latency, high-throughput garbage collection. In: **Proceedings of the 43rd ACM SIGPLAN international conference on programming language design and implementation**. [S. l.: s. n.], 2022. p. 76–91.