Investigation and Optimization of StringDeduplication with Custom Heuristic in Different Versions of the JVM

D. Noetzold, A. G. D. M. Rossetto, J. L. V. Barbosa, and V. R. Q. Leithardt, *Senior Member, IEEE*

***Abstract*—Memory optimization in Java applications is essential for performance and scalability. This paper investigates the efficiency of the *StringDeduplication* parameter in JVM versions 11, 17, and 21, using a *Web Crawler* developed in *Spring Boot*. The results show that the efficiency of *StringDeduplication* decreased from 34.3% deduplication in version 11 to 3.4% in version 21, with an increase in deduplication time from 1,264 ms to 3,439 ms. To mitigate this problem, a custom solution in C was developed for JVM version 21, which increased deduplication to 31.1% and saved 110.2 MB of memory. The main scientific contribution of this work is the identification of the loss of efficiency of *StringDeduplication* in the latest JVM versions and the proposal of a custom solution that improves *String Deduplication*, offering a viable alternative for developers and software engineers.**

Link to graphical and video abstracts, and to code: https://latamt.ieeer9.org/index.php/transactions/article/view/9188

***Index Terms*—*StringDeduplication*, JVM Performance, Memory Optimization, Heuristics, Native Code Integration**

I. Introdução

A otimização do uso de memória em aplicações Java é um tópico de relevância na engenharia de software, devido ao impacto direto no desempenho e na escalabilidade das aplicações [1]. Uma das técnicas introduzidas pela Máquina Virtual Java (JVM) para melhorar a eficiência do uso de memória é a deduplicação de strings. Disponível desde a versão 8u20 do JDK, o *StringDeduplication* é um recurso que permite à JVM identificar e eliminar strings duplicadas na heap, compartilhando instâncias únicas de strings idênticas e, assim, economizando memória [2]–[5]. É importante destacar que a JVM abordada neste artigo é a JVM da Oracle.

O processo de deduplicação de strings na JVM é realizado durante as fases de coleta de lixo (garbage collection). A JVM inspeciona as strings na heap, identifica duplicatas e ajusta as referências para que todas as instâncias idênticas apontem para uma única string [6]. Este mecanismo não apenas reduz o consumo de memória, mas também pode melhorar a performance de aplicações que fazem uso intensivo de strings, como web crawlers, sistemas de processamento de texto e aplicações financeiras [7].

Apesar das vantagens teóricas da deduplicação de strings, a eficácia prática deste recurso pode variar entre diferentes versões da JVM. Com a evolução constante da JVM e a introdução de novas funcionalidades e melhorias, é fundamental investigar como o *String Deduplication* se comporta em versões mais recentes, como 11, 17 e 21 do JDK [2].

Esta pesquisa propõe uma investigação sobre a eficiência do *StringDeduplication* nas versões recentes da JVM. A principal contribuição científica deste trabalho é a criação de uma heurística customizada em C para melhorar a deduplicação de strings na versão 21 da JVM, demonstrando uma significativa melhoria em comparação com a deduplicação nativa.

A heurística personalizada foi desenvolvida para identificar padrões específicos de duplicação de strings e aplicar técnicas para a remoção dessas duplicatas. A implementação da heurística envolveu a análise dos processos de alocação de memória e a integração com os mecanismos de garbage collection da JVM. Essa abordagem conseguiu aumentar a deduplicação de strings para 31.1% e economizar 110.2 MB de memória, oferecendo uma alternativa viável para desenvolvedores e engenheiros de software [8].

O artigo está estruturado da seguinte maneira: a Seção II apresenta os trabalhos relacionados, destacando as pesquisas e trabalhos existentes sobre deduplicação de strings e otimização de memória em JVM. A Seção III detalha a metodologia empregada, incluindo a configuração do ambiente de testes, as ferramentas utilizadas e a implementação da heurística customizada. A Seção IV apresenta os resultados obtidos, com uma análise detalhada das métricas de desempenho e uso de memória. Finalmente, a Seção V conclui o trabalho, destacando as contribuições científicas, as limitações da pesquisa e as direções futuras para pesquisa.

II. Trabalhos Relacionados

Os trabalhos relacionados foram selecionados através de uma busca nas seguintes bases de dados acadêmicas: IEEE Xplore, ACM Digital Library, e Google Scholar. Foram utilizadas as palavras-chave ”String Deduplication”, ”JVM performance”, ”memory optimization”, ”garbage collection” e ”String Deduplication JVM performance”.

No artigo descrito por Horie et al. (2014), onde os autores exploram diversas técnicas de otimização de memória,

The associate editor coordinating the review of this manuscript and approving it for publication was Carlos Thomaz (*Corresponding author: Darlan Noetzold*).

D. Noetzold, and J. L. V. Barbosa are with the University of Vale do Rio Sinos, São Leopoldo, Brazil (e-mails: dnoetzold@edu.unisinos.br, and jbarbosa@unisinos.br).

A. G. D. M. Rossetto is with the Federal Institute Sul-rio-grandense Câmpus Passo Fundo, Passo Fundo, Brazil (e-mail: anubisrossetto@ifsul.edu.br).

V. R. Q. Leithardt is with the Instituto Universitário de Lisboa, Portugal (e-mail: valderi.leithardt@iscte-iul.pt).

incluindo a deduplicação de strings. O artigo destaca que o *String Deduplication* pode reduzir significativamente o uso de memória em aplicações que utilizam um grande volume de strings. No entanto, o trabalho não aborda em detalhes as variações de eficiência entre diferentes versões da JVM [9].

Outro trabalho relevante é o de Nasartschuk et al. (2016), que foca na aplicação da deduplicação de dados no contexto e influência com o Garbage Collector. Embora a pesquisa realizada seja abrangente, ele não fornece uma análise detalhada do desempenho do *String Deduplication* em diferentes versões da JVM [10].

Deng et al. (2017) investigaram a deduplicação de memória como uma abordagem eficaz para melhorar o sistema de memória. Seu artigo concluiu que a deduplicação de memória superou significativamente a compressão de memória em termos de desempenho, destacando a importância da deduplicação em sistemas com grande quantidade de dados repetitivos. Este trabalho reforça a eficácia da deduplicação de dados, mas não foca especificamente na deduplicação de strings em diferentes versões da JVM [11].

Xu et al. (2016) apresentaram o MHDeS, um sistema de deduplicação para gráficos de *Method Handles* no contexto da JVM. A pesquisa demonstrou que o uso de deduplicação de *Method Handles* pode reduzir o consumo de memória em aplicações dinâmicas e melhorar o desempenho da compilação JIT. Embora a abordagem trate de deduplicação, o foco não é diretamente nas strings, mas em gráficos de *handles* de métodos, apresentando uma contribuição relevante para a área de otimização de memória na JVM [12].

Nasartschuk et al. (2016), em outro estudo, propuseram a deduplicação de strings durante a fase de coleta de lixo da JVM. O objetivo era melhorar a gestão automática de memória e reduzir o tamanho do *heap*. Este trabalho foca na interação entre a coleta de lixo e a deduplicação de strings, investigando como esta técnica pode impactar diretamente o número de ciclos de coleta de lixo e, consequentemente, o desempenho da aplicação [13].

A revisão da literatura revela que, embora existam pesquisas positivas quanto a eficácia do *String Deduplication*, há uma carência de pesquisas que avaliem a evolução da eficiência deste parâmetro nas versões mais recentes da JVM.

Este trabalho visa preencher essa lacuna, fornecendo uma análise detalhada da eficiência do *String Deduplication* ao identificar possíveis deteriorações ou melhorias na eficiência do parâmetro. Este artigo contribui para uma compreensão mais aprofundada de como a deduplicação de strings evolui com a JVM e oferecerá uma solução prática para as versões que carecem de uma deduplicação eficiente.

III. Metodologia

Para conduzir os testes e obter os resultados, foi configurado um web crawler em Spring Boot para realizar buscas abrangentes na Wikipedia, nas versões 11, 17 e 21 da JVM com e sem o parâmetro *StringDeduplication*. As métricas foram coletadas utilizando as seguintes ferramentas: Postman, Java Mission Control (JMC) e Java Flight Recorder (JFR). Além disso, foi criado uma biblioteca de deduplicação de strings usando JNI, detalhada a seguir.

A. Aplicações Desenvolvidas

As aplicações desenvolvidas para testar a eficiência do parâmetro *StringDeduplication* nas diferentes versões da JVM, incluindo um web crawler e uma biblioteca de deduplicação de strings utilizando JNI [14].

1) Web Crawler: O desenvolvimento da aplicação de web crawler foi utilizado para testar a eficiência do parâmetro *StringDeduplication* nas diferentes versões da JVM. A aplicação foi desenvolvida em Spring Boot e é responsável por buscar informações em sites com grande hierarquia, como a Wikipedia, a Amazon, o Youtube e o IMDB [15]. Um web crawler funciona automatizando a navegação por páginas da web, seguindo links a partir de uma URL inicial e coletando dados de texto. Ele faz solicitações HTTP às páginas, processa o conteúdo, extrai informações e continua percorrendo o site em busca de novos links para explorar.

a) Execução nas Diferentes Versões da JVM: A aplicação foi executada nas versões 11, 17 e 21 da JVM, mantendo o mesmo código-fonte e ambiente de teste. Isso permitiu uma comparação direta da eficiência do parâmetro *StringDeduplication* entre estas versões, garantindo que qualquer diferença observada nos resultados fosse devida às mudanças na JVM e não a variações no código da aplicação.

As versões 11, 17 e 21 da JVM foram escolhidas para esta pesquisa por serem versões de Long-Term Support (LTS) e por refletirem pontos de evolução na plataforma. A versão 11 foi a primeira LTS após a versão 8, introduzindo o G1 Garbage Collector como padrão, além de melhorias no uso de memória e no desempenho geral da deduplicação de strings. A versão 17 trouxe coletores de lixo adicionais como o ZGC e o Shenandoah, que modificam o comportamento de gerenciamento de memória. A versão 21, sendo a mais recente LTS, reflete as evoluções contínuas em otimizações de coleta de lixo. Avaliar o StringDeduplication nessas três versões permite uma análise das mudanças de eficiência em diferentes momentos de aprimoramento da JVM [2], [5].

2) Biblioteca de Deduplicação para JNI: A biblioteca de deduplicação foi desenvolvida para testar a eficiência do parâmetro *StringDeduplication* através do uso de código nativo C [16], [17]. A seguir, é apresentada uma síntese do funcionamento da biblioteca e os trechos de código mais importantes.

a) Heurística de Deduplicação: A heurística implementada utiliza uma tabela hash para armazenar e verificar a existência de strings duplicadas, sendo eficiente para operações de inserção e busca. A complexidade temporal média de inserção e busca em uma tabela hash é $O(1)$, assumindo uma função de hash eficiente e gerenciamento adequado de colisões. A fórmula que comprova a eficiência da tabela hash é: $T(n) = \frac{n}{m} + O(1)$.

Onde n é o número de entradas e m o tamanho da tabela. Isso significa que, em média, o tempo de busca e inserção permanece constante, mesmo com o aumento de entradas, pois $\frac{n}{m}$ representa a carga da tabela.

A heurística baseada em tabela hash é mais eficiente que a utilizada nas versões 11, 17 e 21 da JVM porque, enquanto o G1 Garbage Collector deduplica as strings de forma reativa durante as fases de coleta de lixo, o uso da tabela hash permite uma deduplicação proativa. Isso significa que a detecção de

strings duplicadas ocorre imediatamente durante a inserção, evitando o acúmulo de duplicatas na memória até a próxima coleta de lixo. Assim, o uso de memória é otimizado em tempo real e o tempo de resposta em aplicações que geram muitas strings é significativamente reduzido, além de garantir um custo constante $O(1)$ para inserções e buscas, em contraste com a sobrecarga adicional do garbage collector.

Algorithm 1 Implementação da heurística de deduplicação usando uma tabela hash.

```
1:  HASH(key)
2:  hashValue ← 0
3:  for i ← 0 to length(key) do
4:      hashValue ← (hashValue ≪ 5) + key[i]
5:  end for
6:  return hashValue%TABLE_SIZE
7:
8:  INSERT(hashMap, key)
9:  slot ← HASH(key)
10: entry ← hashMap.entries[slot]
11: while entry ≠ NULL do
12:     if entry.key = key then
13:         return
14:     end if
15:     entry ← entry.next
16: end while
17: entry ← allocate memory for new Entry
18: if entry = NULL then
19:     print "Failed to allocate memory for Entry"
20:     return
21: end if
22: entry.key ← STRDUP(key)
23: if entry.key = NULL then
24:     print "Failed to duplicate string"
25:     free(entry)
26:     return
27: end if
28: entry.next ← hashMap.entries[slot]
29: hashMap.entries[slot] ← entry
```

O código apresentado no Algoritmo 1 ilustra a implementação da heurística de deduplicação de strings. A função hash (linhas 1-8) calcula o valor de hash para uma dada string, utilizando uma combinação dos valores dos caracteres da string e deslocamento de bits. Este valor de hash é então utilizado para determinar a posição da string na tabela hash.

A função *insert* (linhas 10-24) insere uma nova string na tabela hash. Primeiro, ela calcula o valor de hash da string e encontra o slot correspondente na tabela hash. Em seguida, ela percorre a lista de entradas no slot para verificar se a string já está presente. Se a string já existe, a função retorna sem inserir a duplicata. Caso contrário, a função aloca memória para uma nova entrada, armazena a string e atualiza a lista de entradas no slot.

b) Deduplicação de Strings: A função *deduplicateStrings* utiliza a tabela hash para inspecionar cada string, inserir novas strings não duplicadas e contabilizar as strings deduplicadas. O uso da tabela hash otimiza a busca e inserção, tornando o processo de deduplicação mais eficiente em termos de tempo e memória.

A complexidade de tempo para inserir ou buscar uma string na tabela hash é $O(1)$ em média, comparado a $O(n)$ em uma busca linear simples. Isso resulta em uma melhoria significativa na eficiência do processo de deduplicação.

Para quantificar essa melhoria, considere que o tempo total T para deduplicar n strings usando busca linear é dado por $T_{\text{linear}} = O(n^2)$. Enquanto que, utilizando uma tabela hash, o tempo total T é $T_{\text{hash}} = O(n)$. Essa diferença em complexidade temporal demonstra que a deduplicação usando tabelas hash é significativamente mais rápida, especialmente para grandes conjuntos de strings [18]. A heurística de deduplicação funciona da seguinte maneira: 1. Para cada string s na lista de strings, calcula-se o valor do hash $h(s) = \sum_{i=0}^{k-1} s[i] \times 31^{k-i-1}$. onde $s[i]$ é o valor do caractere na posição i e k é o comprimento da string.

2. Verifica-se se o hash $h(s)$ já está presente na tabela hash: Se $h(s)$ estiver presente, a string é considerada duplicada. Se $h(s)$ não estiver presente, a string é inserida na tabela hash.

3. Mantém-se uma contagem das strings inspecionadas, deduplicadas e o tamanho total deduplicado.

Algorithm 2 Implementação da função de deduplicação de strings usando uma tabela hash.

```
1:  DEDUPLICATE_STRINGS(strings, length,
    inspectedCount, deduplicatedCount,
    deduplicatedSize)
2:  hashMap ← CREATEHASHMAP()
3:  if hashMap = NULL then
4:      print "Failed to create HashMap"
5:      return
6:  end if
7:  *inspectedCount ← 0, *deduplicatedCount ←
    0, *deduplicatedSize ← 0
8:  for i ← 0 to length do
9:      (*inspectedCount) + +
10:     if not CONTAINS(hashMap, strings[i]) then
11:         INSERT(hashMap, strings[i])
12:         (*deduplicatedSize) ← (*deduplicatedSize) +
            STRLEN(strings[i])
13:     else
14:         FREE(strings[i])
15:         strings[i] ← NULL
16:         (*deduplicatedCount) + +
17:     end if
18: end for
```

O código apresentado no Algoritmo 2 ilustra a implementação da função *deduplicateStrings*. A função recebe como parâmetros um array de strings, o comprimento do array, e ponteiros para armazenar as contagens de strings inspecionadas, deduplicadas e o tamanho total deduplicado.

- A função inicializa a tabela hash chamando *createHashMap()* e define as contagens iniciais como zero (linhas 5-8).

- Em seguida, itera sobre cada string no array (linhas 10-22). Para cada string, incrementa o contador de strings inspecionadas (linha 12).
- A função *contains* é utilizada para verificar se a string já está presente na tabela hash (linha 13). Se a string não estiver presente, ela é inserida na tabela hash utilizando a função *insert* e o tamanho deduplicado é atualizado (linhas 14-16).
- Caso contrário, a string é considerada duplicada, é liberada da memória e o contador de strings deduplicadas é incrementado (linhas 17-21).

B. Configuração das JVMs

Para cada teste, as JVMs foram configuradas com o parâmetro *StringDeduplication* ativado e todos os logs de Garbage Collection (GC) habilitados. As configurações específicas das JVMs incluíram:

- Ativação do *StringDeduplication*: *-XX:+UseStringDeduplication*
- Logs de GC: *-Xlog:gc**

C. Captura de Métricas

A captura de métricas foi realizada utilizando o JMC e o JFR durante dez minutos de execução. Os períodos maiores de execução acabaram por ter o mesmo padrão de resultados. As principais métricas analisadas incluíram: uso de memória da heap, número de strings deduplicadas (N_d) [19], número de strings inspecionadas (N_i) [19], tamanho deduplicado (S_d), porcentagem deduplicada (P_d) e tempo de deduplicação (T_d).

A configuração do Java Flight Recorder (JFR) para capturar as métricas necessárias foi realizada com o seguinte comando: *java -XX:StartFlightRecording=duration=1h, filename=myrecording.jfr*. Este comando inicia uma gravação do JFR com duração de 1 hora.

Os logs de GC foram analisados para extrair as métricas relacionadas às strings deduplicadas. A análise dessas linhas permitiu a extração dos valores de N_d, S_d, e T_d, que foram utilizados para calcular P_d e outras métricas de desempenho.

IV. Resultados Experimentais e Análise

Esta seção apresenta os resultados dos testes sobre a eficiência do parâmetro *StringDeduplication* nas versões 11, 17 e 21 da JVM, com métricas como o número de strings inspecionadas, deduplicadas, tamanho e porcentagem deduplicada, e o tempo de deduplicação. Os resultados entre as versões da JVM são comparados para avaliar a eficácia do *StringDeduplication* em um período de dez minutos.

A. Resultados das Métricas de Deduplicação

A Tabela I apresenta os resultados das principais métricas. Observa-se uma clara tendência de diminuição na eficiência do *String Deduplication* à medida que a JVM evolui para versões mais recentes.

Na versão 11, um total de 1,530,837 strings foram inspecionadas, das quais 1,312,865 foram deduplicadas, resultando em uma economia de 101.4 MB de memória e uma porcentagem deduplicada de 34.3%. Nas versões seguintes o número de strings inspecionadas e deduplicadas cai assim como a memória economizada.A versão 21 apresenta a menor eficiência, com apenas 650,245 strings inspecionadas e 299,261 strings deduplicadas. A memória reduz-se para 15.81 MB, correspondendo a uma porcentagem deduplicada de apenas 3.4%.

TABLE I
RESULTADOS DAS MÉTRICAS DE DEDUPLICAÇÃO

Métrica	Java 11	Java 17	Java 21
Strings Inspecionadas	1,530,837	992,693	650,245
Strings Deduplicadas	1,312,865	501,376	299,261
Tamanho Deduplicado	101.4 MB	39.57 MB	15.81 MB
Porcentagem Deduplicada	34.3%	9.3%	3.4%
Tempo de Deduplicação	1,264.442 ms	2,323.492 ms	3,440.466 ms

B. Uso de Memória

Os gráficos de uso de memória foram gerados para comparar o impacto do *StringDeduplication* nas diferentes versões da JVM.

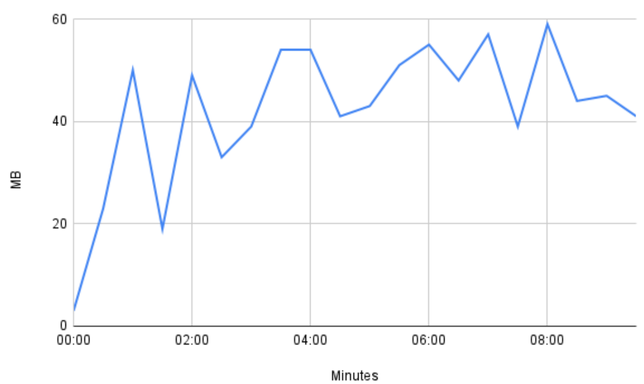

Fig. 1. Uso de memória na JVM 11 com *StringDeduplication*

A Fig. 1 mostra que o uso de memória na JVM 11 com o *StringDeduplication* ativado apresenta uma série de picos e quedas, indicando a atividade de coleta de lixo (GC). A memória utilizada atinge picos de aproximadamente 60 MB antes de ser coletada pelo GC.

A Fig. 2 mostra o uso de memória na JVM 17 com o *StringDeduplication* ativado. Comparado à versão 11, os picos de memória são mais espaçados e menos pronunciados, sugerindo uma eficiência reduzida na coleta de lixo e deduplicação de strings.

A Fig. 3 apresenta o uso de memória na JVM 21 com o *StringDeduplication* ativado mostra picos ainda menos frequentes e menores em comparação com as versões anteriores. Isso pode indicar que a eficácia da deduplicação de strings diminuiu significativamente na versão mais recente.

Sem o StringDeduplication, as versões 11, 17 e 21 da JVM apresentam picos de uso de memória significativamente mais

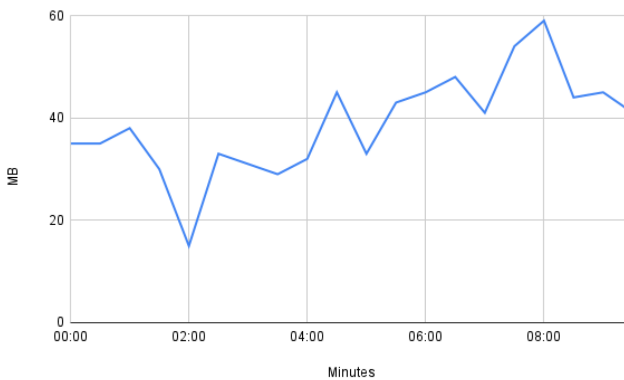

Fig. 2. Uso de memória na JVM 17 com *StringDeduplication*

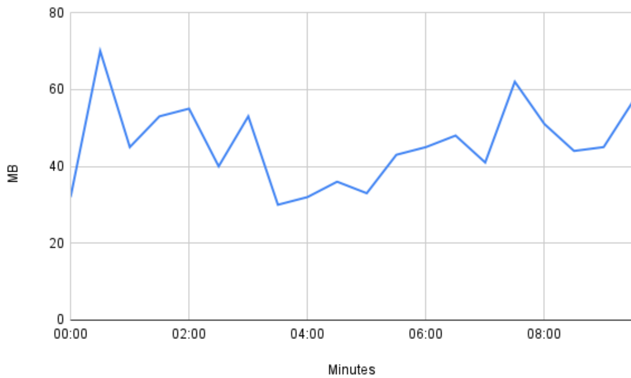

Fig. 3. Uso de memória na JVM 21 com *StringDeduplication*

altos e frequentes, atingindo até 80 MB. Esses resultados confirmam a eficácia do StringDeduplication na redução do uso de memória. Detalhes completos e gráficos podem ser verificados no repositório do GitHub [20].

C. Uso de CPU

Os testes de uso de CPU compararam o impacto do *StringDeduplication* em diferentes versões da JVM. Na JVM 11, com a funcionalidade ativada, o uso de CPU foi moderado, com poucos picos. Na JVM 17, houve um aumento na frequência e altura dos picos, sugerindo maior consumo de recursos. Na JVM 21, os picos foram ainda mais intensos, indicando um consumo significativo de CPU. Sem o *StringDeduplication*, todas as versões (11, 17 e 21) apresentaram menor uso de CPU e menos picos, confirmando que o *StringDeduplication* aumenta a carga de CPU. Detalhes e gráficos estão disponíveis no GitHub [20].

D. Resultados com a Biblioteca criada na JVM 21

Para validar a eficácia da nova heurística de *StringDeduplication*, os testes foram executados na JVM 21, e os resultados foram comparados com os obtidos anteriormente usando o *StringDeduplication* nativo da JVM.

1) Análise dos Resultados: A nova heurística de *StringDeduplication* mostrou melhorias em comparação com a versão nativa da JVM 21, o que pode ser visto na Tabela II. A seguir, destacam-se os principais pontos de comparação:

TABLE II
COMPARAÇÃO DOS RESULTADOS DE DEDUPLICAÇÃO

Métrica	JVM 11	JVM 21 Nativa	JVM 21 com Heurística
Strings Inspecionadas	1,540,837	650,245	1,532,812
Strings Deduplicadas	1,342,865	299,261	1,252,191
Tamanho Deduplicado	102.4 MB	15.81 MB	110.2 MB
Porcentagem Deduplicada	34.3%	3.4%	31.1%
Tempo de Deduplicação	1,264.442 ms	3,439.466 ms	2,421.492 ms

- **Strings Inspecionadas:** A nova heurística inspecionou 1,532,812 strings, um aumento em relação às 650,245 strings inspecionadas pela versão nativa da JVM 21.
- **Strings Deduplicadas:** Foram deduplicadas 1,252,191 strings com a nova heurística, em comparação com 299,261 strings deduplicadas pela versão nativa.
- **Tamanho Deduplicado:** A nova heurística economizou 110.2 MB de memória, um aumento significativo em relação aos 15.81 MB economizados pela versão nativa.
- **Porcentagem Deduplicada:** A porcentagem de deduplicação foi de 31.1%, em comparação com 3.4% da versão nativa.
- **Tempo de Deduplicação:** O tempo de deduplicação foi reduzido para 2,421.492 ms, em comparação com os 3,439.466 ms da versão nativa.

Apesar das melhorias observadas, alguns pontos negativos foram identificados nos resultados:

- **Uso de CPU:** Embora a nova heurística tenha melhorado o tempo de deduplicação, o uso de CPU permaneceu alto em alguns momentos, conforme mostrado na Fig. 4.
- **Portabilidade:** A solução em C pode não ser facilmente portável entre diferentes sistemas operacionais sem ajustes específicos.

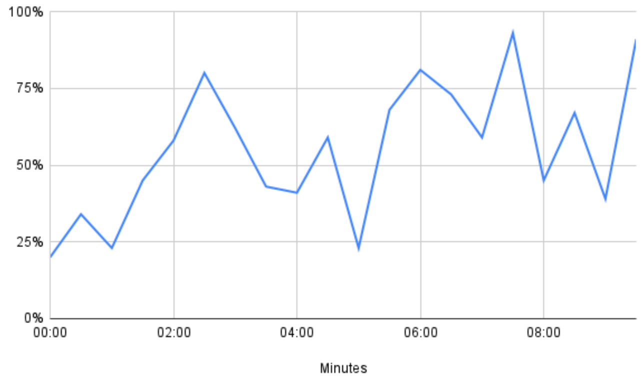

Fig. 4. Uso de CPU na JVM 21 com a nova heurística de *StringDeduplication*

2) Uso de memória: Nesta subseção, discutimos o impacto da nova heurística de deduplicação no uso de memória, comparando os resultados com a versão nativa da JVM 21.

O gráfico de uso de memória na Fig. 5 mostra que a nova heurística de deduplicação conseguiu manter o uso de memória mais estável e eficiente em comparação com a versão nativa da JVM 21. Houve uma redução clara nos picos de memória, o que indica uma melhor gestão das strings duplicadas.

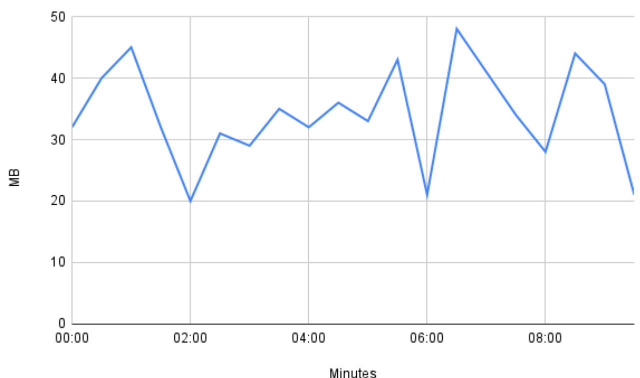

Fig. 5. Uso de memória na JVM 21 com a nova heurística de *StringDeduplication*

E. Discussão dos Resultados

Os resultados indicam que, embora o StringDeduplication ainda ofereça benefícios em termos de economia de memória, sua eficiência está diminuindo nas versões mais recentes da JVM. A análise das métricas revela que o número de strings inspecionadas e deduplicadas diminui significativamente nas versões mais recentes da JVM. Na versão 11, foram inspecionadas 1,540,837 strings, das quais 1,342,865 foram deduplicadas, resultando em uma porcentagem deduplicada de 34.3%. Em contrapartida, na versão 21, foram inspecionadas 650,245 strings, com apenas 299,261 sendo deduplicadas, resultando em uma porcentagem deduplicada de 3.4%.

O tamanho deduplicado também mostra uma redução considerável, com a versão 11 economizando 102.4 MB de memória, enquanto a versão 21 economizou apenas 15.81 MB. Essa diminuição na eficiência da deduplicação sugere que o algoritmo pode estar enfrentando dificuldades em lidar com as mudanças e otimizações das versões mais recentes da JVM.

Além disso, o tempo de deduplicação aumentou de 1,264.442 ms na versão 11 para 3,439.466 ms na versão 21, indicando que o processo está se tornando mais oneroso em termos de tempo de processamento. Este aumento no tempo de deduplicação pode afetar negativamente o desempenho geral da aplicação, especialmente em cenários de alta carga.

Os gráficos de uso de memória demonstram que o *StringDeduplication* reduz significativamente o uso de memória nas versões 11 e 17, mas essa redução é menos pronunciada na versão 21. Na Fig. 1, o uso de memória na JVM 11 com *StringDeduplication* mostra picos de aproximadamente 70 MB, enquanto a Fig. 3 mostra picos menores e menos frequentes na versão 21.

Com a implementação da heurística otimizada de *StringDeduplication* em C, houve uma melhora nos resultados da JVM 21. Foram inspecionadas 1,532,812 strings, das quais 1,252,191 foram deduplicadas, resultando em uma porcentagem deduplicada de 31.1% e economizando 110.2 MB de memória. O tempo de deduplicação foi de 2,421.492 ms.

A diminuição da eficiência do *String Deduplication* nas versões mais recentes da JVM pode ser atribuída a várias otimizações, como o uso de Compact Strings a partir da versão 15, que reduz a quantidade das execuções da deduplicação por conta da compactação de binários [21]. Além disso, coletores de lixo como o ZGC e o Shenandoah (que surgiram na versão 17 da JVM) priorizam a redução de pausas, deduplicando strings com menos frequência para melhorar o desempenho de CPU [22].

Essas mudanças equilibram o uso de memória e performance, mas resultam em menor impacto da deduplicação em comparação às versões anteriores. Importante salientar que a estratégia de *String Deduplication* da JVM Oracle continua a mesma em todas as versões, o que muda é o contexto de aplicação do algoritmo. Esses resultados destacam a eficácia da nova heurística em comparação com a versão nativa do *StringDeduplication* na JVM 21, todos os resultados podem ser visualizados no repositório do projeto no GitHub [20].

V. Considerações Finais

Os resultados indicam que, embora o *StringDeduplication* ainda ofereça benefícios em termos de economia de memória, sua eficácia diminui significativamente nas versões mais recentes da JVM. A implementação da heurística otimizada de *StringDeduplication* em C para a JVM 21 apresentou uma melhoria significativa nos resultados. Porém, os resultados apresentados destacam a necessidade de investigações adicionais para entender as causas da diminuição da eficiência do *StringDeduplication* nas versões mais recentes da JVM. Trabalhos futuros podem explorar os seguintes temas:

- **Impacto em Diferentes Tipos de Aplicações:** Avaliar o impacto do *StringDeduplication* em diferentes tipos de aplicações para determinar se certas categorias de aplicações são mais afetadas pela diminuição da eficiência.
- **Otimizações de Configuração:** Explorar diferentes configurações de JVM e parâmetros de execução para otimizar a eficiência do *StringDeduplication* em cenários específicos.

Este estudo apresenta algumas limitações que devem ser consideradas. Primeiramente, os testes foram realizados em um ambiente controlado, o que pode não refletir os desafios enfrentados em ambientes de produção. Além disso, focamos na comparação do *String Deduplication* nas versões 11, 17 e 21 da JVM, sem explorar o comportamento do algoritmo em uma variedade maior de aplicações. Outra limitação é a ausência de análise de impacto em outros coletores de lixo modernos, como o ZGC e o Shenandoah, que têm características de otimização distintas. Trabalhos futuros podem ampliar essas investigações, fornecendo uma visão mais abrangente sobre a eficácia da deduplicação de strings.

VI. Agradecimentos

A realização desta investigação foi parcialmente financiada por fundos nacionais através da FCT - Fundação para a Ciência e Tecnologia, I.P. no âmbito dos projetos UIDB/04466/2020 e UIDP/04466/2020.

References

[1] A. Goel, C. Prabha, P. Sharma, N. Mittal, and V. Mittal, "Emerging research trends in data deduplication: A bibliometric analysis from 2010 to 2023," *Archives of Computational Methods in Engineering*, pp. 1–18, 2024. DOI: 10.1007/s11831-024-10074-x.

[2] P. Ramya and C. Sundar, "Secdedoop: Secure deduplication with access control of big data in the hdfs/hadoop environment," *Big Data*, vol. 8, no. 2, pp. 147–163, 2020. DOI: 10.1089/big.2019.0120.

[3] M. Basso, A. Rosà, L. Omini, and W. Binder, "Java vector api: Benchmarking and performance analysis," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pp. 1–12, 2023. DOI: 10.1145/3578360.3580265.

[4] C.-Y. Su, A. Bansal, V. Jain, S. Ghanavati, and C. McMillan, "A language model of java methods with train/test deduplication," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 2152–2156, 2023. DOI: 10.1145/3611643.3613090.

[5] X. Yang, R. Lu, J. Shao, X. Tang, and A. A. Ghorbani, "Achieving efficient secure deduplication with user-defined access control in cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 1, pp. 591–606, 2020. DOI: 10.1109/TDSC.2020.2987793.

[6] P. M. A. Kumar, E. Pugazhendhi, and R. K. Nayak, "Cloud storage performance improvement using deduplication and compression techniques," in *Proceedings of the 2022 4th International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pp. 443–449, 2022. DOI: 10.1109/ICSSIT53264.2022.9716524.

[7] S. Xu, D. Bremner, and D. Heidinga, "Mhdes: Deduplicating method handle graphs for efficient dynamic jvm language implementations," in *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pp. 1–10, 2016. DOI: 10.1145/3012408.3012412.

[8] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, "Coverage-based debloating for java bytecode," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–34, 2023. DOI: 10.1145/3546948.

[9] M. Horie, K. Ogata, K. Kawachiya, and T. Onodera, "String deduplication for java-based middleware in virtualized environments," *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 177–188, 2014. DOI: 10.1145/2674025.2576210.

[10] K. Nasartschuk, M. Dombrowski, T. Basa, M. Rahman, K. Kent, and G. Dueck, "Garcosim: A framework for automated memory management research and evaluation," *EAI Endorsed Transactions on Scalable Information Systems*, vol. 3, no. 9, pp. e4–e4, 2016. DOI: 10.4108/eai.14-12-2015.2262678.

[11] Y. Deng, X. Huang, L. Song, Y. Zhou, and F. Z. Wang, "Memory deduplication: An effective approach to improve the memory system," *Journal of Information Science and Engineering*, vol. 33, no. 5, pp. 1103–1120, 2017. DOI: 10.6688/JISE.2017.33.5.1.

[12] S. Xu, D. Bremner, and D. Heidinga, *MHDeS: deduplicating method handle graphs for efficient dynamic JVM language implementations*. New York, NY, USA: Association for Computing Machinery, 2016. DOI: 10.1145/3012408.3012412.

[13] K. Nasartschuk, M. Dombrowski, K. B. Kent, A. Micic, D. Henshall, and C. Gracie, "String deduplication during garbage collection in virtual machines," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, CASCON '16, (USA), p. 250–256, IBM Corp., 2016. DOI: 10.5555/3049877.3049904.

[14] M. Thelwall, "A web crawler design for data mining," *Journal of Information Science*, vol. 27, no. 5, pp. 319–325, 2001. DOI: 10.1177/016555150102700503.

[15] Y. Gao and et al., "Reinforcement learning based web crawler detection for diversity and dynamics," *Neurocomputing*, vol. 520, pp. 115–128, 2023. DOI: 10.1016/j.neucom.2022.11.059.

[16] M. Hirzel and R. Grimm, "Jeannie: Granting java native interface developers their wishes," *ACM Sigplan Notices*, vol. 42, no. 10, pp. 19–38, 2007. DOI: 10.1145/1297105.1297030.

[17] M. Grichi, M. Abidi, F. Jaafar, E. E. Eghan, and B. Adams, "On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (jni)," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 428–440, 2020. DOI: 10.1109/TR.2020.3024873.

[18] H. Shin, D. Koo, and J. Hur, "Secure and efficient hybrid data deduplication in edge computing," *ACM Transactions on Internet Technology (TOIT)*, vol. 22, no. 3, pp. 1–25, 2022. DOI: 10.1145/3537675.

[19] R. Smith and S. Rixner, "Leveraging managed runtime systems to build, analyze, and optimize memory graphs," *ACM SIGPLAN Notices*, vol. 51, no. 7, pp. 131–143, 2016. DOI: 10.1145/3007611.2892253.

[20] D. Noetzold, "String deduplication validation." https://github.com/DarlanNoetzold/StringDeduplicationValidation. Accessed on: August 01, 2024.

[21] P. Boldi and S. Vigna, "Mutable strings in java: design, implementation and lightweight text-search algorithms," *Science of Computer Programming*, vol. 54, no. 1, pp. 3–23, 2005. Principles and Practice of Prgramming in Java (PPPJ 2003).

[22] I. Chaudhary, B. P. Lohani, P. K. Kushwaha, A. D. Gupta, B. Sharma, and J. Sharma, "Generational zgc- an improvement in garbage collector in java 21," in *2024 International Conference on Communication, Computer Sciences and Engineering (IC3SE)*, pp. 631–636, 2024. 10.1109/IC3SE62002.2024.10593533.

D. Noetzold is doing his master's degree at the University of Vale do Rio Sinos (UNISINOS), São Leopoldo, Brazil. He received the degree Bachelor's degree in Computer Science from the Federal Institute Sul-rio-grandense (IFSUL), Passo Fundo, Brazil in 2023. He also works as a Software Developer at CWI Software. His main research interests are High Performance Computing, Ambient Intelligence, and Machine Learning.

A. G. D. M. Rossetto received a Ph.D. degree in Computer Science from the Federal University of Rio Grande do Sul UFRGS/RS, in 2016. Master in Computer Science from the Federal University of Santa Catarina (2007). She is currently a Professor at the Federal Institute Sul-rio-grandense Câmpus Passo Fundo. She maintains cooperation with other research groups in Brazil, France and Portugal. Her main line of research is in distributed systems, mobile computing, internet of things and technologies in education.

J. L. V. Barbosa received M.Sc. and Ph.D. in computer science from the Federal University of Rio Grande do Sul, Brazil. He conducted post-doctoral studies at Sungkyunkwan University (SKKU, Suwon, South Korea) and University of California Irvine (UCI, Irvine, USA). Jorge is a full professor at the Applied Computing Graduate Program (PPGCA) of the University of Vale do Rio dos Sinos (UNISINOS), head of the university's Mobile Computing Lab (MOBILAB), and a researcher at the Brazilian Council for Scientific and Technological Development (CNPq). His main research interests are Ubiquitous Computing, Ambient Intelligence, Big Data, Internet of Things (IoT), and Machine Learning.

V. R. Q. Leithardt (Senior Member, IEEE) received a Ph.D. degree in computer science from INF-UFRGS, Brazil, in 2015. He is currently a Professor with the Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal. His main research interests include distributed systems, focusing on data privacy, communication, and programming protocols, involving scenarios and applications for the Internet of Things, smart cities,big data, cloud computing, and blockchain.