Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Gabriel Alves Tavares

A Serverless-Optimized Garbage Collector

Campina Grande-PB

2025

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

A Serverless-Optimized Garbage Collector

Gabriel Alves Tavares

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Metodologia e Técnicas da Computação

Thiago Emmanuel Pereira da Cunha Silva
(Orientador)

Campina Grande, Paraíba, Brasil

T231s Tavares, Gabriel Alves.
A Serverless-Optimized Garbage Collector / Gabriel Alves Tavares. – Campina Grande, 2025.
59 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2025.
"Orientação: Prof. Dr. Thiago Emmanuel Pereira da Cunha Silva".
Referências.

1. Garbage Collector (GC). 2. Serverless-Optimized Garbage Collector (SOGC). 3. Gerenciamento de Memória - Coletor de Lixo. 4. Serverless. 5. Aplicações de Function-as-a-Service (FaaS). I. Silva, Thiago Emmanuel Pereira da Cunha. II. Título.

CDU 004.43(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA MEIRE EMANUELA DA SILVA MELO CRB-15/568

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secpg@computacao.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

GABRIEL ALVES TAVARES

A SERVERLESS-OPTIMIZED GARBAGE COLLECTOR

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 03/02/2025

Prof. Dr. **Thiago Emmanuel Pereira da Cunha Silva**, UFCG, Orientador

Prof. Dr. **Francisco Vilar Brasileiro**, UFCG, Examinador Interno

Prof. Dr. **Ruan Delgado Gomes**, IFPB, Examinador Externo

Documento assinado eletronicamente por **THIAGO EMMANUEL PEREIRA DA CUNHA SILVA**, **PROFESSOR 3 GRAU**, em 24/03/2025, às 12:04, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Ruan Delgado Gomes**, **Usuário Externo**, em 24/03/2025, às 14:35, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **FRANCISCO VILAR BRASILEIRO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 25/03/2025, às 10:16, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **5325665** e o código CRC **1977745E**.

Referência: Processo nº 23096.002456/2025-54 SEI nº 5325665

Resumo

Linguagens de programação gerenciadas abstraem o gerenciamento de memória de baixo nível, permitindo que programadores se concentrem em questões de alto nível. Nessas linguagens de programação, o Gerenciamento Automático de Memória (GAM), frequentemente implementado por meio de um Coletor de Lixo (GC, do inglês, Garbage Collector), lida automaticamente com a alocação e desalocação de memória. Embora o GAM mitigue erros relacionados à memória, sua sobrecarga pode impactar o desempenho da aplicação. GCs modernos empregam técnicas como concorrência, gerenciamento de memória generacional e algoritmos adaptativos para minimizar esse impacto no desempenho.

Esta dissertação se concentra em estratégias de GC especificamente adaptadas para aplicações de Function-as-a-Service (FaaS). As cargas de trabalho de FaaS exibem um padrão distinto de uso de memória, caracterizado por objetos efêmeros e persistentes. Diferentemente das abordagens tradicionais de GC de propósito geral, neste trabalho, propomos e avaliamos um novo algoritmo de GC, o Serverless-Optimized Garbage Collector (SOGC). O SOGC aproveita as características únicas do FaaS para obter ganhos significativos de eficiência.

Um ciclo típico de uso de memória em uma função FaaS envolve uma fase de inicialização, durante a qual os dados destinados a toda a vida útil da função são alocados, seguida por uma fase de processamento de eventos, caracterizada por dados efêmeros que são usados para processar o evento e, em seguida, descartados rapidamente. O SOGC aborda esse padrão organizando a memória em um layout que inclui um espaço persistente para dados de longa duração e um espaço de processamento separado para cada evento. Essa organização eficiente da memória permite uma rápida recuperação de dados não utilizados, minimizando interrupções relacionadas ao coletor de lixo durante a execução da lógica de negócios.

Para avaliar o SOGC, empregamos um modelo analítico, permitindo uma comparação direta com algoritmos GC clássicos. Por meio desse modelo, avaliamos vários cenários, demonstrando que o SOGC tem o potencial de superar as soluções existentes sob certas condições.

Abstract

Managed programming languages abstract away low-level memory management, enabling programmers to focus on high-level concerns. In these programming languages, Automatic Memory Management (AMM), often realized through a Garbage Collector (GC), automatically handles memory allocation and deallocation. While AMM mitigates memory-related errors, its overhead can impact application performance. Modern GCs employ techniques such as concurrency, generational memory management, and adaptive algorithms to minimize this performance impact.

This dissertation focus on GC strategies specifically tailored for Function-as-a-Service (FaaS) applications. FaaS workloads exhibit a distinct memory usage pattern, characterized by ephemeral and persistent objects. Unlike traditional, general purpose GC approaches, in this work we propose and evaluate a novel GC algorithm, Serverless-Optimized Garbage Collector (SOGC). SOGC takes advantage of FaaS unique characteristics to achieve significant efficiency gains.

A typical memory usage cycle in a FaaS function involves an initialization phase, during which data intended for the function's entire lifetime is allocated, followed by an event handling phase, characterized by ephemeral data that is used to process the event and then promptly discarded. SOGC addresses this pattern by organizing memory into a layout that includes a persistent space for long-lived data and a separate handler space for each event. This efficient memory organization allows for rapid reclamation of unused data, minimizing garbage collector-related interruptions during the execution of business logic.

To evaluate SOGC, we employ an analytical model, enabling a direct comparison with classic GC algorithms. Through this model, we assess various scenarios, demonstrating that SOGC has the potential to outperform existing solutions under certain conditions.

Agradecimentos

Esta dissertação é resultado do esforço conjunto de toda a rede de apoio e colegas de trabalho que tive nesta jornada. Eu não teria conseguido finalizá-la sem o incentivo de meus amigos, professores e família durante esses mais de dois anos de pesquisa.

Primeiramente, agradeço ao meu orientador, Thiago Emmanuel, ou Manel, pela paciência e dedicação em me ajudar a sempre seguir em frente e dar um passo de cada vez. Suas dicas e orientação foram extremamente valiosas e levo-as comigo para além do ambiente acadêmico ou da indústria, mas também para a vida. Além disso, meu não oficial coorientador, Daniel Fireman, foi essencial para minha pesquisa e desenvolvimento profissional. Agradeço por todas as conversas e o precioso feedback que sempre me foi dado. Graças a essa ajuda, entendi o verdadeiro significado do dito: "O maior produto de um mestrado não é a pesquisa, e sim o pesquisador".

Agradeço também ao amor da minha vida, Thayla. Ela acompanhou pessoalmente as incontáveis noites em que eu trabalhava enquanto descobria, aos poucos, o quão difícil é escrever. Seu imenso apoio, carinho e paciência me ajudaram a sempre seguir em frente e entender meu próprio ritmo. Também sou grato aos meus amigos, tanto dos laboratórios da UFCG como de fora, que assistiram às minhas apresentações e revisaram minhas ideias, contribuindo muito para minha pesquisa.

Por último, agradeço à minha família, principalmente minha mãe e minha irmã, que, mesmo de longe, me apoiam nas minhas aventuras do outro lado do país.

O presente trabalho foi realizado com apoio e financiamento do projeto VTEX Acelera/EMBRAPII e da Bolsa 09/2022 do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

Contents

Lista de Símbolos

AMM - *Automatic Memory Management*

GC - *Garbage Collector*

FaaS - *Function-as-a-Service*

SOGC - *Serverless Optimized Garbage Collector*

M&S - *Mark and Sweep*

List of Figures

List of Tables

List of Source Codes

Chapter 1

Introduction

Managed programming languages are prevalent in modern software development, powering a wide range of services and applications. Their popularity is due to their ability to simplify the implementation process, allowing programmers to focus on higher-level problem-solving and abstractions. These languages achieve this by delegating the complexities of operating system interactions to the language runtime, effectively abstracting away the intricate details of resource management. In contrast, traditional programming languages often require developers to explicitly manage the memory allocation of even the simplest data structures, thus demanding more cognitive effort.

Automatic Memory Management (AMM) is a cornerstone of managed programming languages. AMM significantly simplifies development by handling memory allocation and deallocation automatically, eliminating the need for explicit programmer intervention. In languages like Java or Python, developers simply declare variables and data structures, and the language runtime handles memory management transparently. In most cases, developers do not need to worry about explicitly freeing memory – the language runtime takes care of this automatically.

AMM is often implemented by the Garbage Collector (GC), which is responsible for memory management. Despite its name, a GC's duties extend beyond freeing memory. A comprehensive GC must, at a minimum, perform three essential processes: allocation, identification, and reclamation. Specifically, the GC is tasked with:

- **Allocation:** Managing the allocation of all objects and data structures in the heap section of memory;

- **Identification:** Identifying memory occupied by unreachable objects (those not in use by the application);
- **Reclamation:** Safely removing these unreachable or "dead" objects to free memory for future allocations.

Although a robust GC greatly simplifies development, it does come with trade-offs. The concurrent execution of AMM background processes can impact the performance of the mutator – which is the program running within the managed environment, responsible for requesting memory allocation [11]. This contention for resources, inherent in all AMM algorithms, can cause pauses or delays in application execution, often referred to as garbage collection pauses.

Significant efforts have been devoted to mitigating the impact of garbage collection (GC) on mutator performance. For example, some approaches focus on implementing more efficient concurrent GC operations, aiming to reduce interference with application execution, as demonstrated by Shenandoah [18] for the Java Virtual Machine or using techniques such as parallel garbage collection for shared memory multiprocessors [7], while others leverage generational memory management, a common strategy used to focus collection on objects more likely to be reclaimed, exemplified by the Garbage-First (G1) garbage collector [10]. Region-based memory management [25] also offers an alternative approach that uses memory regions to bound the overhead of collection, which can be relevant in scenarios where memory locality is important. These techniques, along with others like Immix [4], a mark-region garbage collector, show that garbage collectors can significantly enhance application performance when matched to the appropriate environment, configuration, and application requirements. All of these advances and many others are related to the idea of better managing the heap and reducing the performance overhead associated with it.

This dissertation introduces a novel GC algorithm specifically designed for the Function-as-a-Service (FaaS) programming model. FaaS has emerged as a popular approach for developing and deploying cloud-based applications [12], abstracting away infrastructure complexities and allowing developers to focus on writing code that responds to events triggered by user interactions. FaaS finds applications across diverse domains, including web applications and machine learning pipelines [20, 6, 13]. These diverse applications, while varying in their

specific functionalities, share a common characteristic: they rely on managed programming languages, where memory management and thus garbage collection is a core component of their runtime environment.

Memory management in FaaS is both a cost (resource usage) and performance (latency) issue, due to FaaS billing model. Poorly configured GCs can increase resource consumption, directly translating to higher costs. While unoptimized resource usage is detrimental in any application, it is further exacerbated in serverless [1] environments due to their latency-sensitive and event-driven nature; FaaS providers bill clients by the time each function is invoked. Additionally, serverless functions are frequently deployed with constrained resource configurations, making them more susceptible to the performance impacts of GC. Some alternatives even avoid part of the GC overhead by starting runtimes from a pre-warmed GC checkpoint [23, 15]. Chapter 2 explores how FaaS functions utilize memory and how GCs can leverage the unique characteristics of this programming model.

Our proposal, The Serverless-Optimized garbage collector (SOGC) strategically organizes memory into two distinct regions: a persistent space for data that persists across function invocations, and a handler space that is allocated for each new event. This approach allows SOGC to quickly reclaim memory used during each invocation by simply discarding the handler space after the function's execution, minimizing the overhead of traditional garbage collection and aiming to reduce interruptions during the execution of the function's business logic.

In summary, the objective of this work is to propose a low-overhead GC algorithm for serverless functions that is not tied to a specific runtime and directly addresses the performance and cost challenges of Automatic Memory Management in FaaS environments. In this direction, in the following chapters, we present the following contributions:

1. We review automatic memory management in the context of serverless by analyzing the unique characteristics of FaaS memory usage (Chapter 2);

2. We present the design of the Serverless-Optimized Garbage Collector (SOGC): a new GC algorithm for FaaS that leverages the properties of FaaS functions to improve

[1] The term Function-as-a-Service is often used synonymously with "serverless", since FaaS is the most prevalent use case for serverless computing. We use the terms FaaS and Serverless as synonyms in our work.

memory management (Chapter 4);

3. We present an analytical evaluation of garbage collectors in serverless environments, establishing a model and a framework for comparing different algorithms (Chapter 5);

4. We analyze the performance of different garbage collectors using our analytical model, comparing the proposed SOGC algorithm with traditional approaches in different scenarios (Chapter 6).

Finally, in Chapter 7, we present conclusions and future work derived from this dissertation.

Chapter 2

Context

This chapter lays the foundation for understanding the challenges and opportunities in optimizing garbage collection for Function-as-a-Service (FaaS) environments. We begin by exploring the core principles of the FaaS model, examining how applications within this paradigm utilize memory resources. Following this, we overview Automatic Memory Management (AMM) and its operational mechanics, providing the essential context for understanding the design choices that led to our algorithm.

2.1 The Function-as-a-Service Offering

Function-as-a-Service (FaaS) has emerged as a popular model for developing and deploying cloud-based applications [12]. FaaS abstracts away the complexities of the underlying infrastructure, enabling developers to focus solely on writing code. FaaS has been used to build applications across diverse domains, including web applications and machine learning pipelines [20, 6, 13]

FaaS functions are event-driven, ensuring that resources are allocated only when needed, reducing idle time and associated costs. Functions are triggered by events such as HTTP requests, database updates, or file uploads. This event-driven nature enables elasticity, where functions can automatically scale resources up or down based on demand.

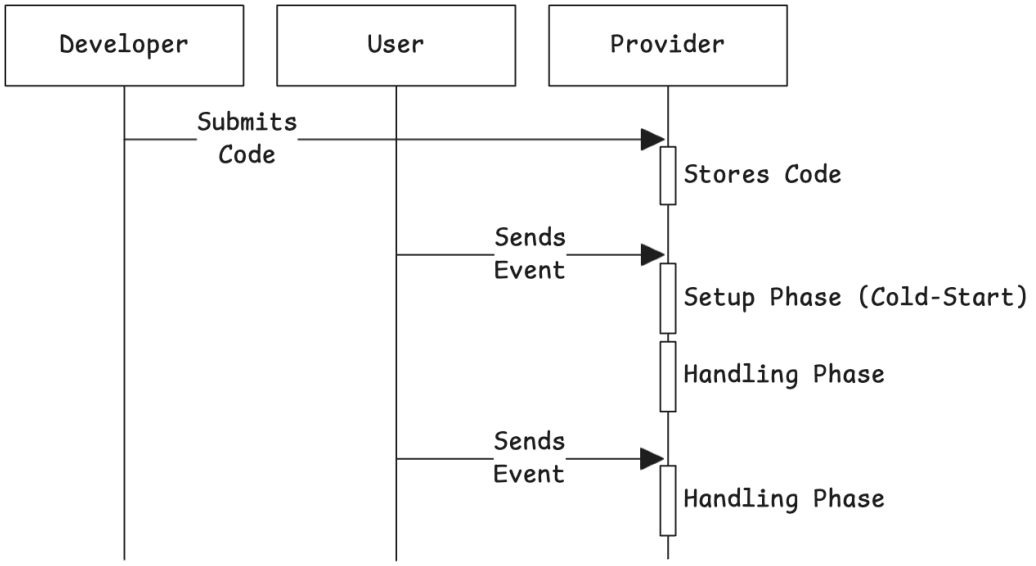

Figure 2.1: Sequence diagram illustrating the FaaS function lifecycle. The diagram shows a *Developer* submitting code to the *Provider*, followed by a *User* triggering the function with an event. The *Provider* then initiates a *Setup Phase*, often referred to as the cold-start, before executing a *Handling Phase* to process the event. Subsequent events are handled directly via the execution of the handling phase.

2.1.1 Memory allocation on FaaS Applications

Consider the code depicted in Source Code 2.1. This code could represent a user-provided function within a FaaS environment, tasked with processing images by compressing them and subsequently storing them in a database. This is the core application logic, or the business logic, for the user's specific task. However, for this code to function correctly, it requires dependencies from the FaaS environment. These dependencies include establishing a connection to the database, setting up an event listener, opening HTTP connections, and potentially loading necessary libraries.

```
1 def handle_image_event(event, context):
2     image_data = event.get('image_data')
3     if not image_data:
4         return { 'statusCode': 400 }
5     decoded_image = base64.b64decode(image_data)
6     storage_location = store_image(decode_image)
```

```
7     return { 'statusCode': 200 }
```

Source Code 2.1: Example of a FaaS function in Python. This code illustrates at high-level a image compression process.

As illustrated in Figure 2.1, after the user submits the function code, the FaaS provider awaits incoming events. The first event triggers a setup phase, during which the environment, including cloud resources and runtime dependencies, is loaded. Immediately following the setup, the event is handled by the user's code. In our example, this would involve compressing and storing the image. Subsequent events bypass the setup phase, being directly processed by the user's code in the handling phase.

This process highlights a clear distinction between persistent memory usage and ephemeral memory usage, a key aspect that influences how memory is managed in FaaS environments. Persistent memory, such as the HTTP connections that should remain open and the database connection, persists across multiple invocations and requires careful management to prevent resource exhaustion. Conversely, ephemeral memory, used in the handling phase for variables holding the raw and compressed images, becomes garbage and is eligible for disposal after the handling phase is completed.

The need to efficiently manage this ephemeral memory, alongside the persistent resources, introduces a level of complexity that is typically addressed by Automatic Memory Management (AMM). Understanding the intricacies of AMM is crucial for comprehending the challenges inherent in managing memory within FaaS environments.

2.2 Automatic Memory Management

In traditional programming languages, such as C and C++, the programmer is responsible for memory management, explicitly allocating and deallocating memory. This programmer-led memory management is error-prone, often leading to memory leaks, dangling pointers, crashes, and undefined behavior. These issues make it hard to develop and maintain applications.

To address this challenge, Automatic Memory Management (AMM) was developed, shifting the responsibility of memory management from the programmer to the system.

AMM performs several crucial tasks to ensure that memory is used effectively and safely. The first of these responsibilities is the dynamic **allocation** of objects in the Heap section of memory. When the mutator needs a new object, the GC finds a suitable memory space in the Heap and marks it as occupied. The allocation triggers the GC to manage the memory using, for example, strategies that find the next available memory space. Second, AMM involves the **identification** of unreachable objects, that is, finding memory that is no longer in use. Over time, the Heap will fill up with objects that are no longer referenced. The GC must identify these objects to reclaim them. Third and last, AMM is responsible for the **reclamation** of unused memory from the Heap. Once garbage has been identified, the GC needs to free the memory associated with the related objects. To reclaim the memory, GCs usually defragment the heap, grouping the available memory, and addressing the fragmentation caused by the allocation and deallocation of objects over time.

To illustrate the concepts of allocation, identification, and reclamation, and to understand how garbage collection operates, we can refer to Figure 2.2, which depicts a simplified memory model. In this figure, the boxes on the left represent different scopes of execution, analogous to stack frames in an operating system. The *Main Scope* represents the application's main execution context, which persists throughout the entire process. The other scopes, *Function Scope 1* and *Function Scope 2*, represent different executions of the `handle_image_event` function. In the FaaS setting, the *Main Scope* corresponds to the runtime environment, while each function scope represents a specific invocation of the image compression code.

Within these scopes, we have root nodes—memory pointers referencing locations within the heap, where objects are dynamically allocated. When a scope terminates, these pointers become eligible for reclamation, because they are intrinsically tied to the scope's lifetime, and the memory they refer to in the stack is released. However, these root nodes often point to objects in the *Heap*, which can, in turn, reference other objects, forming a chain of references. To determine which objects are still in use and which are eligible for reclamation, a garbage collector needs to perform a reachability analysis. One common way to do this—and the one we present here for simplicity—is by tracing all the pointers, starting from root nodes in active scopes and following all the references to the heap objects. Objects that are reachable are considered live, and everything else is considered garbage and can be reclaimed.

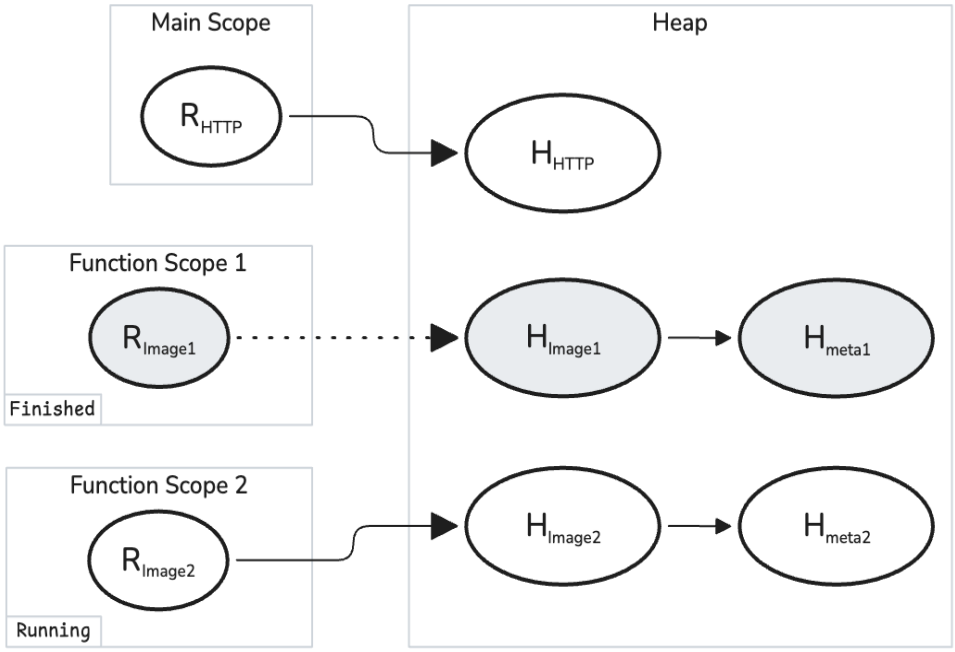

Figure 2.2: Example of memory scopes, object reachability, and lifetime. The figure shows three distinct scopes: Main Scope, Function Scope 1, and Function Scope 2, with their respective root nodes (R_{HTTP}, R_{Image1}, and R_{Image2}). The Heap represents the dynamic memory area where objects are allocated. Heap nodes H_{Image1} and H_{meta1} are allocated in Function Scope 1, which has finished, making them unreachable. Heap node H_{HTTP} is allocated in the Main Scope. Heap nodes H_{Image2} and H_{meta2} are allocated in Function Scope 2, which is currently running.

Let's map these concepts to the example in Figure 2.2. The *Main Scope* has a root node, R_{HTTP}, which points to a heap object, H_{HTTP}. This object holds information about the connection, such as its type, version, and open ports, and remains live throughout the function's lifetime. *Function Scope 1* represents a finished execution of the `handle_image_event` function. Its root node, R_{Image1}, has been discarded with the function's stack frame. Consequently, the nodes containing the image's data, H_{Image1} and H_{meta1}, are no longer reachable from any active scope, and are thus considered garbage. Finally, *Function Scope 2* represents a running execution of the function, with its root node, R_{Image2}, and corresponding references, H_{Image2} and H_{meta2}, still active.

A core challenge for garbage collectors is balancing two conflicting goals: they must identify and collect garbage effectively, yet avoid interfering with the application's execu-

tion. To accomplish this delicate balance, GCs require resources, such as CPU cycles and memory access, leading to overheads inherent in all AMM strategies. When the GC needs to identify unreachable memory, it may pause mutator threads, which effectively halts the execution of all application threads, significantly impacting application responsiveness. These pauses, often referred to as "stop-the-world" pauses, can lead to noticeable latency spikes, particularly in latency-sensitive applications like web services. As highlighted in the seminal work "Tail at Scale" [9], even infrequent high-latency events caused by GC pauses can disproportionately degrade the user experience. For example, if the garbage collector pauses the mutator threads while handling an HTTP request from a user, the time for that request may increase considerably, causing the application to become unresponsive. This is further illustrated in Figure 2.3.

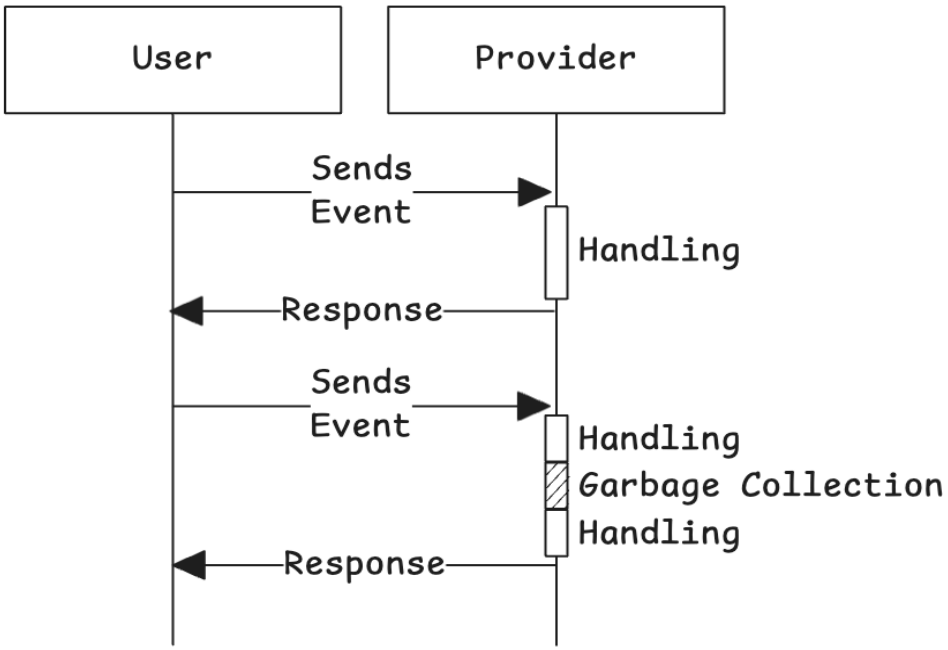

Figure 2.3: Example of interference between garbage collector and mutator execution. The GC impact can manifest itself as a pause in mutator threads during the handling of events, effectively increasing the time to respond queries for the user, thus increasing user-side latency.

Many garbage collection strategies aim to reduce the impact of these pauses. For example, some GCs employ techniques such as generational collection, which segregates objects based on their age to focus the collection efforts on areas that are likely to contain more

garbage [10], and concurrent collection, which overlaps the GC process with application execution, reducing the length of these "stop-the-world" pauses [18]. This study adopts a similar objective by leveraging the unique memory allocation characteristics of serverless functions. Specifically, we propose a segregated memory space to optimize garbage collection for Function-as-a-Service (FaaS) environments, separating persistent and ephemeral memory into distinct regions to simplify and enhance the efficiency of their respective collections.

This new approach to the problem will define a new garbage collection algorithm that leverages the memory usage of FaaS. We will dive deeper into the definition of this new algorithm in Chapter 4.

Chapter 3

Related Work

This chapter discusses previous research that proposed new garbage collection (GC) algorithms, explored improvements in Automatic Memory Management (AMM), and analyzed memory management in Function-as-a-Service (FaaS) applications. This allows us to contextualize our work within the existing body of knowledge and highlight its potential contributions to the serverless paradigm.

3.1 On Creating New Garbage Collectors

The creation of general-purpose GCs is a very complex task that involves trade-offs, and to create a new one, a lot of research should be done to evaluate the impact of each strategy. Modern industry-standard collectors, built as generalists, implement many optimizations to try to perform well under any scenario or application. While these collectors are designed to perform well in most situations, that does not mean that they are the best for specific applications, which creates an opportunity for research into specialized GCs.

Bacon et al. [2] introduced a unified theory of garbage collection that provides a framework for understanding and comparing different algorithms. Their work emphasizes the importance of balancing trade-offs between Mutator Utilization (U), Space Utilization (S), and Collection Cost (C). In our work, we also aim to find a good trade-off between these variables, but with a specific focus on FaaS applications, instead of general-purpose ones.

In 2008, Blackburn et al. introduced Immix, a novel mark-region garbage collector featuring opportunistic defragmentation and a new heap structure [4]. A key innovation of

Immix is its approach to allocating and reclaiming memory in contiguous regions, utilizing coarse blocks where possible and finer-grain lines when necessary, which distinguishes it from simpler tracing GCs. Immix's mark-region tracing strategy sweeps contiguous free regions of the heap, offering potential performance benefits. As a general-purpose garbage collector, Immix aims to perform well across a variety of applications. Consequently, its evaluation, like that of many generalist collectors, typically employs benchmark suites that may not fully reflect the unique characteristics of serverless environments. Our work, in contrast, is specifically targeted towards optimizing garbage collection for the FaaS programming model.

Following Immix, in 2022, Zhao et al. introduced LXR [26]. LXR employs a similar heap structure to Immix, but instead of relying on costly read/write barriers, it implements reference counting, highlighting the fundamental similarities between tracing and reference counting techniques, as described by Bacon et al. [3]. While Immix uses a mark-region approach, LXR leverages reference counting to identify and reclaim dead objects. Like Immix, LXR is a general-purpose collector and its evaluation has primarily focused on benchmarks that do not specifically reflect FaaS-like application characteristics.

Every GC strategy presents its own set of trade-offs, and understanding these limitations is crucial for determining the appropriate use case. For example, Shenandoah, a GC algorithm, prioritizes minimizing pause times to achieve high throughput and low latency [18]. However, previous work suggests that Shenandoah's emphasis on short pauses can lead to a higher frequency of concurrency operations and more expensive read/write barriers, potentially resulting in lower overall performance in certain benchmarks [26].

3.2 Improving AMM on Interactive Cloud Services

Besides proposing new GC algorithms, some studies also focused on improving existing AMM techniques, by acting on the application instead of the garbage collector itself. They made a background task controller that monitors GC behavior and predicts when a garbage collection will happen [14, 22, 16]. This collector-agnostic approach, called Garbage Collector Control Interceptor (GCI), is valuable for cloud services. By predicting when a garbage collection will occur, the system can optimize the load balancing, by rerouting requests to

replicas that are not going to be impacted by a collection. Their experiments show that this technique reduces AMM's impact on the application's long-tail latency.

Our work complements GCI very well by providing an alternative to slow garbage collectors. While GCI reduces the impact on latency, our proposed SOGC algorithm is optimized for fast garbage collections, which also enhances the availability of the service. These two strategies can be combined to create more robust and efficient systems.

3.3 Garbage Collection for Specific Use Cases

Some researchers have explored garbage collection algorithms tailored for specific use cases, providing valuable insights into how GCs can be adapted for different environments and constraints.

- **Real-Time Garbage Collection:** Several works have focused on developing garbage collectors that can provide low latency and predictable pause times for real-time systems. The work of Bacon et al. [3] presents a real-time garbage collector for Java, and the work of Cheng et al. [7] explores, for the first time, the challenges of real-time garbage collection running on shared-memory multiprocessors.

- **Garbage collection for parallel programs:** Several works have explored parallel garbage collection techniques to improve performance in multi-core systems. The work of Appel et al. [1] presents a parallel garbage collection algorithm, and the work of Flood et al. [17] explores the challenges of concurrent garbage collection.

- **Region-Based Memory Management:** Introduced by Tofte and Talpin [25], presents an alternative approach to traditional garbage collection. The core idea is to allocate objects into regions, which are then deallocated as a whole when they are no longer needed. A type and effect system automatically infers where regions can be allocated and deallocated, aiming to provide memory safety without relying on a garbage collector. In the context of FaaS, where minimizing resource consumption and latency is paramount, the concept of region-based memory management could be valuable. However, the overhead of a full static analysis by the compiler to determine these

regions might be excessive for the short-lived and dynamic nature of serverless functions. Furthermore, in FaaS, the separation of persistent and ephemeral data provides a natural partitioning of memory that obviates the need for complex compiler analysis to identify regions. Our serverless-optimized garbage collector, in contrast, leverages this inherent separation, directly addressing the unique memory allocation patterns and lifecycle of FaaS functions without requiring extensive static analysis.

3.4 Serverless Performance Evaluation

To propose improvements for Function-as-a-Service (FaaS) environments, it is necessary to understand how to evaluate new proposals within this paradigm. Several studies have developed FaaS benchmarks that tackle the challenge of testing serverless functions, creating standardized sets of metrics, functions, and tests. The authors in the latest serverless benchmark studies acknowledge that a benchmark for FaaS should encompass a comprehensive list of workloads, representative applications, and tests that address aspects unique to serverless architectures—such as cold-start latency—which differ from traditional ones.

One such study is by Copik et al., who introduced SeBS, an open-source FaaS benchmark offering new metrics and experiments, alongside a representative model for FaaS. This benchmark allows both local and remote tests [8]. Their study demonstrated SeBS's potential to identify discrepancies between billing costs and actual resource usage, compare cold-start latencies across platforms, and measure provider availability. However, the authors did not address the impact of automatic memory management on FaaS performance.

Somu et al. released PanOpticon, a unique serverless benchmark that focuses on displaying metrics related to function chaining and trigger overhead [24]. This benchmark aims to measure a comprehensive set of FaaS provider metrics, helping users automate testing of various options. These options include changing the provider, adjusting the memory assigned to functions, altering the request rate, modifying function chain lengths, and other unique parameters. However, this benchmark does not address automatic memory management strategies, and the available applications lack specific memory access patterns that would be relevant for garbage collection evaluation.

Martins et al. proposed a diverse suite of benchmarks for serverless platforms [21]. The

authors aimed to capture essential aspects specific to serverless, including the overhead of the serverless platform, differences between programming languages within this paradigm, and container reutilization. Their suite included a test that varies the memory allocated per container, but it does not evaluate different memory management strategies and uses only basic "Hello-world" functions for this particular test, which would not stress the application in any significant way.

In summary, state-of-the-art benchmark suites reflect the current priorities of their field. The lack of automatic memory management metrics, evaluation, and memory configuration options in serverless benchmarks demonstrates the relevance of our approach, which proposes and evaluates a new garbage collection algorithm, SOGC, and an analytical model for garbage collection evaluation.

Chapter 4

The Serverless-Optimized Garbage Collector

Our approach to garbage collection is specifically designed for the memory usage patterns of Function-as-a-Service (FaaS) applications. Instead of designing a general-purpose algorithm, we focus on the characteristics of this programming model. In this chapter, we will detail the constraints that define our specific scenario, describing the memory usage patterns we are targeting. We will then present how our algorithm organizes application memory, followed by a description of the algorithm itself. Finally, we will outline the expected benefits of this new approach.

4.1 Constraints

The Serverless-Optimized Garbage Collector (SOGC) is specifically designed to thrive in environments where memory usage exhibits a distinct pattern: a clear separation between long-lived, persistent data and short-lived, ephemeral data associated with individual event executions. SOGC excels when the majority of an application's persistent state is established during a setup or initialization phase, and the bulk of memory allocated during event handling is temporary, becoming irrelevant once the event has been processed. This pattern is particularly well-suited to FaaS applications, where each function invocation is often treated as an independent transaction. While some limited allocation of persistent resources during event handling might be possible, it should be minimized to maintain the efficiency

of SOGC. SOGC needs each event execution to have a handler memory allocation with a limited size.

The power of SOGC lies in its ability to rapidly reclaim the ephemeral memory associated with each event. By assuming that handler memory is, in fact, ephemeral, SOGC can avoid the overhead of traditional garbage collection techniques.

It is important to note that, if an event requires significantly more memory than the allocated handler space, a backup memory management strategy could be employed, similar to how reference counting uses tracing garbage collection to handle circular references. If the memory of some application reaches this state a lot it means that it does not respect our constraints anyway and our algorithm is not suited for it.

4.2 Memory Layout

By following these constraints, memory usage becomes more predictable and manageable, allowing for efficient garbage collection strategies. To take advantage of these constraints, we propose a new memory layout that segregates persistent and temporary resources for more efficient memory management and quick reclamation of resources after each event:

- **Setup Memory:** This pool is defined beforehand and allocated during the setup phase to store all persistent resources of the function, remaining throughout its lifecycle.
- **Handler Memory:** A new memory pool is created for each event handled by the mutator. This pool stores temporary resources and is marked as garbage and reclaimed immediately after the event handling concludes.

The SOGC algorithm, which is designed to efficiently manage memory in serverless environments, leverages the memory layout described above. The process starts by initializing the Setup Memory pool, where all persistent objects are stored, and creating reusable Handler Memory pools. When an event arrives, the algorithm allocates a Handler Memory pool from the set of reusable pools, and then uses this pool to process the event. After processing the event, the Handler Memory is marked as garbage and its resources are reclaimed. This simple reclamation is a direct consequence of the previous constraints, and does not require complex tracing mechanisms. Although these constraints are strict and may not be suitable

for all applications, the FaaS programming model encourages the creation of environments where applications naturally adhere to them.

4.3 Algorithm

Algorithm 1 Serverless-Optimized Garbage Collector (SOGC)

Require: Memory pools for setup and handler phases are not initialized

Ensure: Memory is efficiently managed and reclaimed in a serverless environment

```
1:  procedure INITIALIZE_MEMORY_POOLS
2:      setup_memory ← CREATE_MEMORY_POOL
3:      ALLOCATE_SETUP_OBJECTS(setup_memory)
4:      handler_memory_pools ← CREATE_HANDLER_MEMORY_POOLS
5:  end procedure
6:  procedure HANDLE_EVENT(event)
7:      handler_memory ← ALLOCATE_HANDLER_MEMORY(handler_memory_pools)
8:      PROCESS_EVENT(event, handler_memory)
9:      MARK_HANDLER_MEMORY_AS_GARBAGE(handler_memory)
10:     RECLAIM_HANDLER_MEMORY(handler_memory_pools, handler_memory)
11: end procedure
12: while IS_EVENT_HANDLING do
13:     event ← GET_NEXT_EVENT
14:     HANDLE_EVENT(event)
15: end while
```

The SOGC algorithm is designed to efficiently manage memory in serverless environments by leveraging the previously defined memory layout. The process begins by initializing the Setup Memory pool, where persistent resources are allocated, and by creating a set of reusable Handler Memory pools. Upon each incoming event, the algorithm allocates a new Handler Memory pool from the set of reusable pools. The event is then processed using this pool, and once the event handling is complete, the Handler Memory is immediately marked as garbage and reclaimed. This reclamation is a simple operation, possibly a matter

of resetting a pointer. This ensures that no leftover resources from event handling persist, maintaining efficient memory usage.

The simplicity of this algorithm is a direct result of the constraints we established. Knowing that Handler Memory pools become garbage after each event eliminates the need for complex tracing mechanisms. Additionally, the fixed allocation size for each Handler Memory pool means the algorithm does not need to adapt to varying memory demands across different events, avoiding the need for traditional garbage collection cycles.

While these constraints are strict and may not be suitable for all applications, we believe that the FaaS programming model itself strongly encourages the creation of environments where applications naturally adhere to these rules. This makes SOGC particularly well-suited for FaaS workloads.

4.4 Expected Benefits

Leveraging the specific memory patterns of FaaS, the SOGC design offers several key benefits for serverless environments:

- **Reduced Latency Spikes During Event Handling:** By segregating persistent and temporary resources, SOGC aims to reduce garbage collection-related pauses during event processing.
- **Fast Garbage Collection:** By grouping all garbage within the Handler Memory pool, SOGC allows for a simpler and quicker reclamation process, eliminating the need for complex tracing or sweeping algorithms.
- **Reduced Memory Fragmentation:** By quickly reclaiming entire Handler Memory pools, SOGC minimizes memory fragmentation, ensuring efficient use of memory resources.
- **Simplified Memory Management:** The separation of concerns by using specific memory pools and the single point of reclaim for each pool makes SOGC algorithm easier to understand, implement and optimize.

Chapter 5

Garbage Collection Model

Typically, new garbage collectors are evaluated experimentally using benchmarks [26, 12, 4, 5]. However, this approach presents significant challenges within the scope of our study. Benchmarking requires fully functional implementations of each GC algorithm under consideration, as well as ensuring that these implementations are in comparable states of optimization. Comparing a newly developed algorithm from a single research cycle with mature, industry-proven GCs could lead to unfair conclusions that do not accurately represent the potential of the new algorithm. Furthermore, such an approach would require significant development efforts, which are beyond the scope of this research.

Therefore, we have opted for an analytical approach to evaluate our Serverless-Optimized Garbage Collector (SOGC). Our review of the literature provided us with the foundations to formulate a new analytical model for comparing algorithms without the need for complete, optimized implementations. While analytical models inevitably involve simplifications and constraints, they enable understanding core performance trade-offs while maintaining a useful level of comparability between algorithms. This approach allows us to study the behavior of each algorithm without being influenced by particular implementation details.

Our model predicts the performance of the GC, for a specific collection algorithm subject to the activity of the mutator. To achieve this, we combined the performance model created by Heymann [19] with the memory model discussed by Bacon et al. [2]. Using this combined model, we described three algorithms: 1) Mark-and-Sweep (M&S); 2) Copy-Live-Objects (COPY); and our own SOGC. The model described here is used in the next chapter to compare these algorithms.

5.1 Memory representation and the Garbage Collection Cycle

Our garbage collection process follows Bacon et al. [2] definition of the memory as a graph $G = (V, E)$, where:

- The set of vertices V represents allocated memory nodes;
- The set of directed edges E represents references between objects;
- Root nodes $R \subseteq V$ represents external references;
- The "free list" $F \subseteq V$ of available memory nodes;
- The live vertices $V_L \subseteq V$, those reachable from R;
- Dead vertices $V_D \subseteq V$, defined as $V_D = V - V_L$, which are not reachable from R.

In this model, a *vertex* is a point in the graph G representing a memory location. We will refer to these memory locations as *nodes*. The application stores *objects* in these memory *nodes*. The garbage collector has the duty of reclaiming the memory *nodes* not reachable from the root set, that is storing dead *objects*. We will use the term "node" to refer to the smallest unit of memory that is allocated or reclaimed by a garbage collector, and "object" to refer to the program entity stored in memory.

To maintain a tractable analytical model, we simplify the garbage collection cycle by focusing on a streamlined scenario of allocation followed by reclamation. Our garbage collection cycle uses the following steps:

1. The system begins with a partially filled heap (memory graph) containing live nodes (V_L).
2. New objects are allocated into the memory graph at a defined allocation rate, continuously adding nodes to V and increasing the set of live nodes V_L, until the heap is full ($F = \emptyset$).
3. Once full, the garbage collector is invoked, reclaiming dead nodes (nodes in V_D), making new free space available in F and returning the heap to a state comparable to the beginning of the cycle.

Based on the above blueprint, in the next section we can model the performance of the selected algorithms.

5.2 Garbage Collector Performance

Following Heymann [19], in our performance model we consider that, during the collection cycle, the mutator performs useful computation, which we define as the productive time (τ_p), while the garbage collector manages memory, incurring a garbage collection time overhead (τ_o).

The total cycle time (τ_c), representing the complete duration of a garbage collection cycle, is defined as the sum of the productive time and the garbage collection time overhead:

$$\tau_c = \tau_p + \tau_o \tag{5.1}$$

In the context of our GC cycle, the productive time (τ_p) represents the time it takes to allocate all of the objects which will be garbage in the current cycle. It can be calculated using the **allocation rate** (λ), the rate at which new nodes are used to allocate memory. This parameter directly affects how quickly the memory fills up, impacting the frequency of garbage collections and therefore, the performance of all algorithms. When inverted ($\frac{1}{\lambda}$) it represents the time taken to allocate one memory node. Multiplying by the number of nodes allocated during a cycle, we have the following equation for the productive time:

$$\tau_p = \frac{|V| - |V_L|}{\lambda} \tag{5.2}$$

The garbage collection time overhead (τ_o), on the other hand, is particularly defined for each algorithm, since the overhead depends on the algorithm's design. Each algorithm's overhead will be discussed in the following sections, but common to them are the definitions of **core operation costs** that capture the costs associated with each garbage collection algorithm's core operations. These costs are expressed in terms of time and represent the overhead incurred by the GC. Examples of such costs include:

- The cost c_m to **mark** a single node as live;
- The cost c_s to **scan** a single memory node to find object references;

- The cost c_a to **allocate** a single memory node;
- The cost c_r to **reclaim** a single memory node;
- The cost c_c to **copy** a single live object to another node in memory.

It's important to note that while parameters like the cost to allocate a node (c_a) are common to multiple algorithms, their actual values can vary significantly based on the specific garbage collection algorithm. For instance, Mark-and-Sweep may require additional operations, such as initializing bits to mark the node, making the allocation operation more costly compared to Copy-Live-Objects, which do not require this additional overhead. Similarly, our proposed SOGC algorithm introduces costs related to the initialization of its memory spaces, reflecting its specific memory layout designed for FaaS environments.

Finally, our model includes a **overallocation factor** (ρ), defined as the ratio of the total number of allocated memory nodes ($|V|$) to the number of live memory nodes ($|V_L|$), i.e., $\rho = \frac{|V|}{|V_L|}$. This dimensionless factor represents the proportion of total allocated memory space relative to the memory occupied by live nodes. This ratio is a key parameter when one is configuring any GC in practice, since GCs need more or less space to work properly. As a consequence, discussing performance results in terms of ρ helps system operators' decision-making. For example, operators can better tune their system to a specific scenario (for example, provide more heap memory when needed). Also, it is possible to help operators decide between algorithms on a given setting.

5.2.1 Mark-and-Sweep

The Mark-and-Sweep (M&S) algorithm works by first marking all live memory nodes reachable from the root set, and then sweeping through the entire memory, reclaiming the unmarked nodes. During each cycle, all the live nodes ($|V_L|$) must be marked, each with a cost of c_m. Additionally, all allocated memory nodes ($|V|$) must be scanned, each with a cost of c_s, and the dead nodes ($|V_D| = (|V| - |V_L|)$) must be allocated (c_a) and reclaimed (c_r). Therefore, the total garbage collection time overhead (τ_o) for M&S can be expressed as:

$$\tau_o = |V_L|c_m + |V|c_s + (|V| - |V_L|)(c_a + c_r) \tag{5.3}$$

To express this equation in terms of the overallocation factor (ρ), which is defined as $\rho = \frac{|V|}{|V_L|}$, we can rewrite $|V|$ as $\rho|V_L|$ and substitute into the previous equation:

$$\tau_o = |V_L|c_m + \rho|V_L|c_s + (\rho|V_L| - |V_L|)(c_a + c_r) \tag{5.4}$$

Now, we can factor out $|V_L|$ from the equation:

$$\tau_o = |V_L|(c_m + \rho c_s + (\rho - 1)(c_a + c_r)) \tag{5.5}$$

Finally, we can group terms with ρ:

$$\tau_o = |V_L|(c_m - c_a - c_r + \rho(c_s + c_a + c_r)) \tag{5.6}$$

5.2.2 Copy-Live-Objects

The Copy-Live-Objects algorithm operates by copying all live nodes from one memory region, referred to as the *fromspace*, to another, known as the *tospace*. This action effectively compacts the live data, eliminating any fragmentation that may have accumulated in the *fromspace* and subsequently making the old *fromspace* region available for reuse in subsequent cycles. Within our model, this translates to the algorithm requiring an additional $|V_L|$ space compared to Mark-and-Sweep; this extra space allows the live objects to be copied to new nodes within the tospace region. Put simply, the total memory consists of a space for the live nodes ($|V_L|$), a free space that will receive those live objects upon the invocation of a collection ($V_{L'}$, where $|V_L| = |V_{L'}|$), and the space for nodes allocated during the cycle that will become garbage ($|V_D|$). Consequently, we can express the total memory as:

$$|V| = |V_L| + |V_{L'}| + |V_D| = 2|V_L| + |V_D| \tag{5.7}$$

During each collection cycle, the nodes that will eventually become garbage are implicitly allocated in the *fromspace* (incurring a cost of c_a per node), and each live node must be copied from the *fromspace* to the *tospace*, with a cost of c_c for each. Thus, the total garbage collection time overhead (τ_o) for Copy-Live-Objects can initially be expressed as:

$$\tau_o = |V_L|c_c + |V_D|c_a \tag{5.8}$$

To express this equation in terms of the overallocation factor (ρ), we can rewrite the number of dead nodes $|V_D|$ as $|V| - 2|V_L|$, using equation 5.7. Consequently, the overhead becomes:

$$\tau_o = |V_L|c_c + (|V| - 2|V_L|)c_a \tag{5.9}$$

Using the fact that $\rho = \frac{|V|}{|V_L|}$, we can write $|V|$ as $\rho|V_L|$:

$$\tau_o = |V_L|c_c + (\rho|V_L| - 2|V_L|)c_a \tag{5.10}$$

Finally, factoring out $|V_L|$:

$$\tau_o = |V_L|(c_c + (\rho - 2)c_a) \tag{5.11}$$

5.2.3 Serverless-Optimized Garbage Collector

This algorithm takes into account the specific memory characteristics of FaaS, where there is a setup phase for persistent memory, which by design will not be subject to garbage collection, and a handler phase that allocates nodes which will be garbage. Consequently, the SOGC introduces a specific parameter to capture the unique aspects of its memory layout: the cost for **persistent memory setup** (c_t), which is the cost to set up the memory for the persistent objects. The total garbage collection time overhead (τ_o) for SOGC is calculated as the cost to set up the persistent memory plus the cost to allocate and reclaim the handler memory. This overhead can be initially expressed as:

$$\tau_o = |V_L|c_t + (|V| - |V_L|)(c_a + c_r) \tag{5.12}$$

To express this equation in terms of the overallocation factor (ρ), which is defined as $\rho = \frac{|V|}{|V_L|}$, we can rewrite $(|V| - |V_L|)$ as $(\rho - 1)|V_L|$, and substitute into the previous equation:

$$\tau_o = |V_L|c_t + (\rho - 1)|V_L|(c_a + c_r) \tag{5.13}$$

Now, we can factor out $|V_L|$ from the equation:

$$\tau_o = |V_L|(c_t + (\rho - 1)(c_a + c_r)) \tag{5.14}$$

Chapter 6

Evaluation

This chapter presents the results obtained from our analytical model, which compares the performance of the Mark and Sweep (M&S), Copy-Live-Objects (COPY), and Serverless-Optimized Garbage Collector (SOGC) algorithms across different scenarios.

The performance of the algorithms is evaluated in terms of two key metrics, **Productivity** (π) and **Effectivity** (η), adapted from Heymann[19]. The **Productivity** is the proportion of the time that the application is active, is given by:

$$\pi = \frac{\tau_p}{\tau_c} \tag{6.1}$$

In its turn, **Effectivity** is defined as the productivity divided by the overallocation factor. This metric expresses the overhead and the productivity of each algorithm with relation to the extra space that is required to execute the GC:

$$\eta = \frac{\pi}{\rho} \tag{6.2}$$

The results are organized into distinct scenarios. Each scenario represents a specific configuration of parameters, or a variation of a particular parameter, allowing us to evaluate the behavior of all garbage collectors under different conditions. For instance, one scenario examines the impact of varying the allocation rate, which allows us to assess the performance of each algorithm under different memory pressure conditions.

6.1 Parametrization

To ground our analytical model costs, we map the core operations of the garbage collection algorithms to sequences of assembly instructions, using the instruction latencies from Agner Fog's instruction tables for the Intel Knights Landing architecture [1]. The assembly instructions representing the core operations can be found in the Appendix A.

6.1.1 Mark-and-Sweep

The table below summarizes the core operations of the Mark-and-Sweep garbage collector, the instructions they use, and their calculated latencies.

Operation	**Instructions Used**	**Latency (Cycles)**
Marking	'MOV'	3
Scanning	'MOV', 'CMP', 'JL', 'JG'	3
Allocation	'MOV'	2
Reclamation	'MOV'	2

Table 6.1: Assembly instructions for Mark-and-Sweep operations and their latencies.

Based on these mappings, the parameter values for Mark and Sweep are set as follows:

- Marking (c_m): The marking operation was mapped to three 'MOV' instructions, one for memory read and two for register operations, having a combined cost of 3 cycles.
- Scanning (c_s): The scanning operation was mapped to instructions for memory read with 'MOV', and two comparison operations 'CMP' with the heap start and the heap end, having a combined cost of 3 cycles. Other instructions in this mapping have no cost.
- Allocation (c_a) and Reclamation (c_r): Both allocation and reclamation operations were mapped to 'MOV' instructions and have similar latencies of 2 cycles.

[1] Data available on: https://agner.org/optimize/instruction_tables.pdf

6.1.2 Copy-Live-Objects

The table below summarizes the core operations of the Copy-Live-Objects garbage collector, the instructions they use, and their calculated latencies.

Operation	**Instructions Used**	**Latency (Cycles)**
Copying	‘MOV‘	2
Allocation	‘MOV‘	2

Table 6.2: Assembly instructions for Copy-Live-Objects operations and their latencies.

Based on these mappings, the parameter values for Copy-Live-Objects are set as follows:

- Allocation (c_a): The allocation operation in the Copy-Live-Objects algorithm is mapped to two MOV instructions, and thus has a latency of 2 cycles.
- Copying (c_c): The copying operation involves loading data from memory and storing it again in another memory address. This mapping includes two ‘MOV‘ instructions, thus having a latency of 2 cycles

6.1.3 Serverless-Optimized Garbage Collector

The table below summarizes the core operations of the Serverless-Optimized Garbage Collector, the instructions they use, and their calculated latencies.

Operation	**Instructions Used**	**Latency (Cycles)**
Persistent Memory Setup	‘MOV‘	3
Handler Allocation	‘MOV‘	1
Handler Reclamation	‘MOV‘	1

Table 6.3: Assembly instructions for Serverless-Optimized Garbage Collector operations and their latencies.

Based on these mappings, the parameter values for Serverless-Optimized Garbage Collector are set as follows:

- Persistent Memory Setup (c_t): The persistent memory setup operation was mapped to memory load and data structure initialization operations that use MOV instructions, having a combined cost of 3 cycles.

- Handler Allocation (c_a): The handler allocation operation was mapped to a simple ‘MOV‘ instruction and a single memory access, thus having a cost of 1 cycle.
- Handler Reclamation (c_r): The handler reclamation operation was mapped to a ‘MOV‘ instruction and a single memory access, thus having a cost of 1 cycle.

This approach, which maps core operations to assembly instructions and uses instruction latencies, provides a more realistic foundation for comparing the trade-offs between different garbage collection algorithms. While the parameter values are derived from a simplified mapping and do not account for the complexities of real-world implementations, including microarchitectural effects and the variable cost of memory accesses (e.g., cache and RAM), they offer a useful starting point for our analysis. Future work should focus on validating these parameters with micro-benchmarks and performance analysis in a real environment.

6.2 Base Scenario

We now analyze the performance metrics of each garbage collector based on our model. For a base scenario, we use the parameter values defined in Section 6.1, and these values are summarized in Table 6.4, organized by garbage collector.

Garbage Collector	Parameter	Value
M&S	c_s	1
	c_m	3
	c_a	1
	c_r	1
COPY	c_a	2
	c_c	2
SOGC	c_t	3
	c_a	1
	c_r	1
ALL	λ	0.1

Table 6.4: Parameter values for the base scenario, separated by garbage collector.

6.2.1 Productivity

Figure 6.1 illustrates the productivity of each garbage collection algorithm using the default parameter values defined in Section 6.1. First, as expected, for $\rho < 2$, COPY shows zero productivity. SOGC exhibits higher productivity for most of the range of overallocation. Only for very large overallocation values, COPY shows hints of being slightly better than SOGC. Beyond $\rho = 3.5$, the productivity of all three algorithms starts to converge,

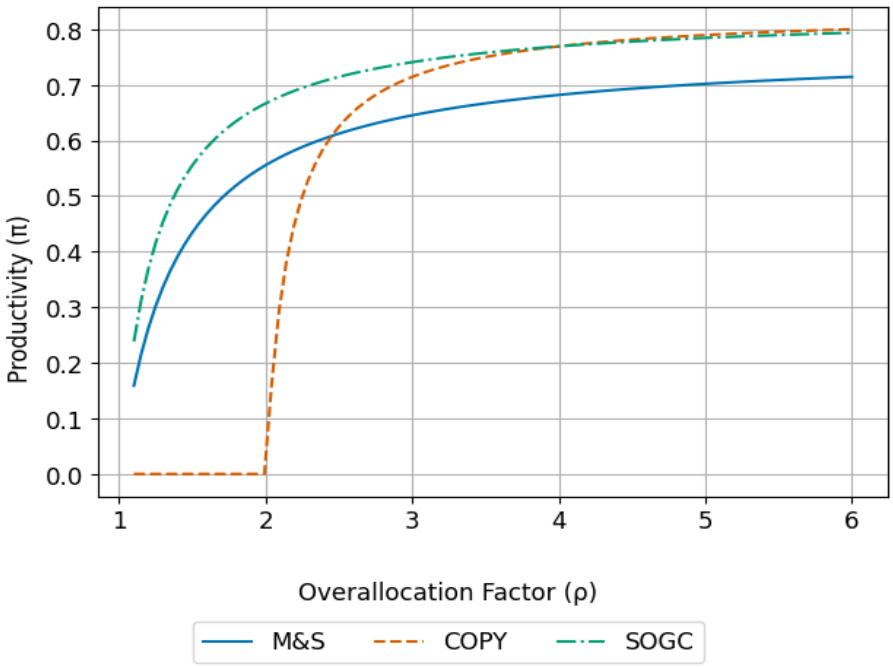

Figure 6.1: Productivity (π) vs. Overallocation Factor (ρ) for M&S, COPY, and SOGC in the base scenario. The plot shows that SOGC achieves higher productivity than M&S for most values of ρ, with COPY surpassing it by a narrow margin only at high overallocation factors ($\rho > 3.7$).

These results suggest that, using our baseline parameters, SOGC's optimized memory management outperforms M&S and maintains higher productivity from low overallocation factors up to approximately $\rho = 3.7$, where its performance converges with COPY. In the context of FaaS applications, lower overallocation factors are particularly relevant because users usually try to configure function memory close to the minimum required, in order to optimize costs. Then, our results indicate that FaaS applications might benefit from the proposed approach.

6.2.2 Effectivity

To further analyze the trade-offs between performance and memory utilization, we now consider the effectivity (η) of the three garbage collection algorithms in the base scenario. Effectivity, defined as the ratio of productivity (π) to the overallocation factor (ρ), provides a measure of how efficiently an algorithm utilizes memory while maintaining performance.

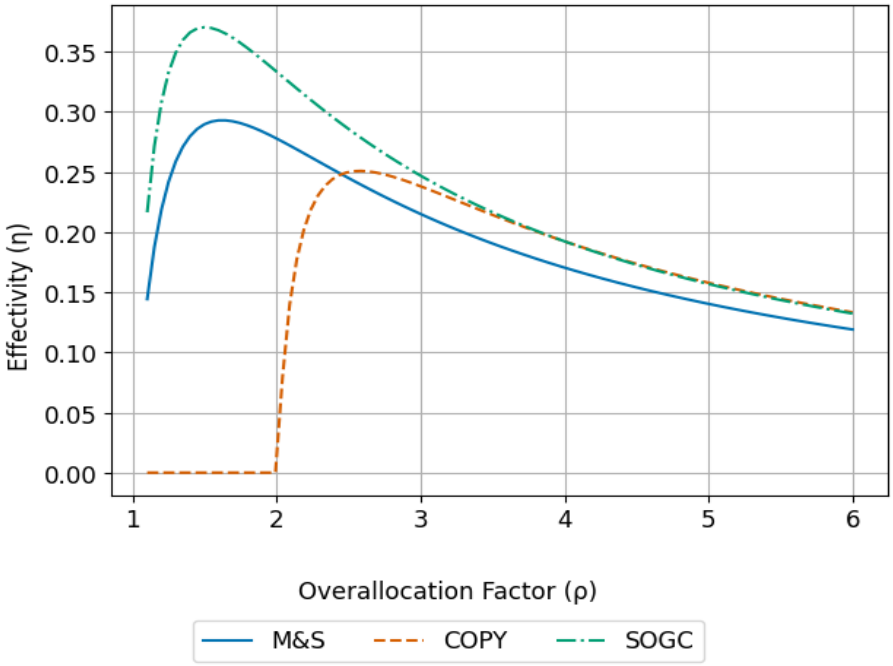

Figure 6.2: Effectivity (η) vs. Overallocation Factor (ρ) for M&S, COPY, and SOGC in the base scenario. The plot shows how SOGC reaches the highest effectivity at very low overallocation factors, and how COPY has a sharp rise after $\rho = 2$, while M&S is lower in all cases.

Figure 6.2 illustrates the effectivity of M&S, COPY, and SOGC across varying values of the overallocation factor (ρ). The plot reveals that SOGC exhibits the highest effectivity at very low overallocation factors, achieving a peak value around $\rho = 1.5$, and outperforming both M&S and COPY. As the overallocation factor increases, the effectivity of both SOGC and M&S declines, while COPY's effectivity rises sharply, starting at $\rho = 2$, as expected for this algorithm. All algorithms exhibit a gradual decline in effectivity for $\rho > 2.7$, but while SOGC and COPY maintain similar levels, M&S is consistently lower.

These results reveal a performance trend mirroring that of the productivity metric; for

instance, the overallocation factor at which COPY surpasses M&S in effectivity aligns with the point where it surpasses M&S in productivity. However, a notable divergence emerges at lower overallocation values. Specifically, at $\rho = 1.5$, SOGC demonstrates an effectivity approximately 1.3 times greater than that of M&S. This, coupled with SOGC's superior productivity within this range, strongly suggests that our algorithm is both faster and more space-efficient in its memory management, a characteristic of particular relevance for FaaS environments where low overallocation is frequently the most critical consideration.

6.3 Productivity in Other Scenarios

In the following sections, we will analyze the results of our model for additional scenarios. It is important to note that the effectivity metric in these scenarios exhibits a trend consistent with the productivity metric, maintaining the relative performance of each garbage collector. Therefore, to avoid redundancy, we will focus our analysis solely on the productivity results for these specific scenarios.

6.3.1 The Impact of Allocation Rate

Figure 6.3 summarizes the impact of varying object allocation rates on the productivity of the three garbage collection algorithms. We have established three distinct levels for the allocation rate: *Low*, *Medium*, and *High*. The *Low* rate is set to be ten times lower than the base scenario, *Medium* corresponds to the base scenario rate, and the *High* rate is ten times higher than the base scenario.

At the *Low* allocation rate, all garbage collectors demonstrate similarly high productivity, which is a direct result of the minimal memory pressure imposed by the system. This outcome serves as a crucial validation for our model, as it aligns with the expectation that garbage collectors should have a negligible impact on application performance when serving a slow workload. As the allocation rate increases to *Medium* and subsequently to *High*, the productivity of all GCs declines, and no particular algorithm exhibits a clear advantage in handling higher allocation rates over the others.

Despite the decrease in absolute productivity, the relative performance of each garbage collector remains consistent across all allocation rates. SOGC continues to outperform the

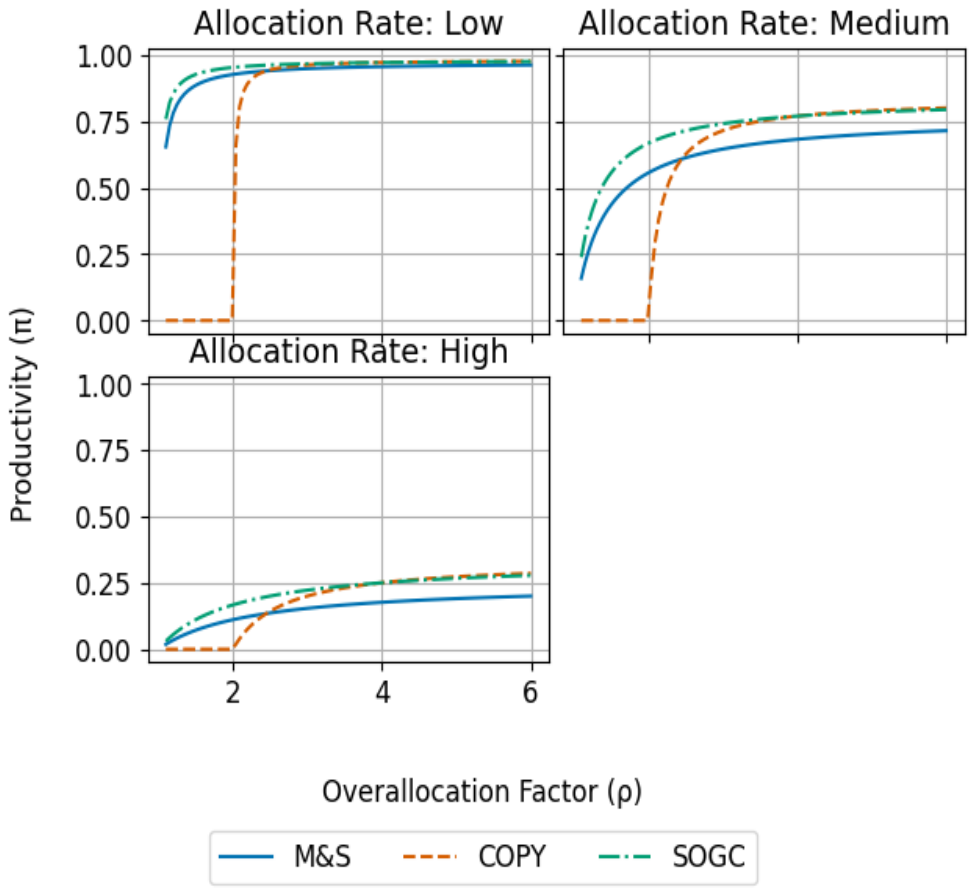

Figure 6.3: Productivity (π) vs. Overallocation Factor (ρ) for M&S, COPY, and SOGC under varying allocation rates. This plot highlights the consistent relative performance of each GC algorithm when the allocation rate changes.

other algorithms for most values of ρ, up to approximately 3.5, and COPY surpasses both M&S and SOGC only in very high overallocation factors. These results show that our proposed SOGC algorithm remains the better option for applications operating in realistic ranges of overallocation factors, even when the allocation rate varies.

In FaaS environments, where functions may experience variable pressure due to fluctuations in user demand, understanding the behavior of garbage collectors under these conditions is paramount for the development of novel algorithms. Therefore, future work should prioritize refining the garbage collector parameterization, exploring the inclusion of concurrency and other complex factors within the model, and subsequently revisiting this scenario to gain more nuanced insights into the algorithms' behavior.

6.3.2 The impact of SOGC optimizations

The SOGC optimization scenario, illustrated in Figure 6.4, explores the impact of varying the setup costs for SOGC. It's important to acknowledge that, beyond the design of a garbage collection algorithm, its implementation plays a critical role in its overall performance. Mature garbage collectors are highly optimized, so this section analyzes the potential impact of implementation improvements, which are reflected as lower parameter costs. To achieve this, this analysis considers different levels of optimization for the persistent memory setup, ranging from an *Optimistic* configuration, where these costs are three times lower than the base scenario, to a *Pessimistic* configuration, where these costs are three times higher. A *Regular* configuration, using our base parameter values, was also tested.

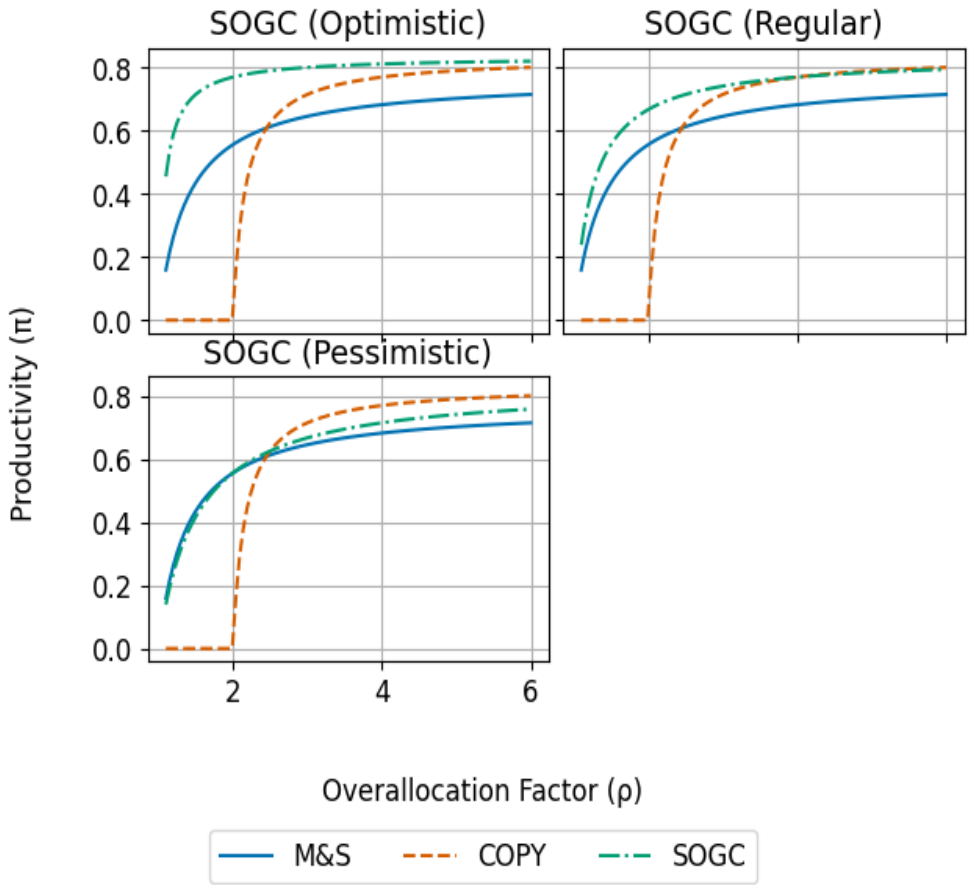

Figure 6.4: Productivity (π) vs. Overallocation Factor (ρ) for M&S, COPY, and SOGC under different SOGC setup costs (Optimistic, Regular, and Pessimistic). This plot highlights the sensitivity of the SOGC algorithm to the configuration of its setup parameters.

The *Optimistic* configuration demonstrates SOGC's potential, showing that it can outperform both M&S and COPY across the entire range of the overallocation factor (ρ). In a better implementation scenario, SOGC can achieve even higher performance than in the base scenario. For instance, within the range $1 < \rho < 4$, SOGC's productivity is up to 1.5 times greater than M&S and also outperforms COPY even at higher values of ρ. Conversely, the *Pessimistic* configuration reveals a substantial performance drop, with SOGC performing similarly to M&S up to $\rho = 2.4$, beyond which COPY becomes the more productive algorithm. This highlights that a poorly optimized setup phase could easily render SOGC less effective than other options for all values of ρ.

We acknowledge that our default modeling of SOGC was inherently pessimistic, as we incorporated a setup cost not present in the other GCs. This decision was deliberate, designed to create a parameter that allows us to gauge the accuracy of our constraints for this particular GC. If it is indeed possible in a real-world scenario to completely separate the setup phase from the collection cycle without incurring additional amortized costs during future collections, then the *Optimistic* scenario would more closely reflect reality. Conversely, if segregating these two memory regions proves difficult and the setup memory requires constant maintenance throughout the cycle, then the *Pessimistic* scenario would be more representative. Ultimately, the implementation of SOGC must prioritize efficient setup and initialization processes, as these aspects significantly influence its overall performance and determine whether the proposed approach proves beneficial or detrimental.

Chapter 7

Conclusion

This work introduced the Serverless-Optimized Garbage Collector (SOGC), a novel garbage collection algorithm designed to improve memory management in Function-as-a-Service (FaaS) environments. We proposed an analytical model, combining the abstract framework proposed by Bacon et al. [2] with a simplified memory cycle and performance metrics inspired by Heymann [19], to evaluate the performance of the proposed algorithm in comparison with traditional garbage collection techniques, namely Mark-and-Sweep (M&S) and Copy-Live-Objects (COPY).

Our results demonstrate that SOGC exhibits higher productivity and effectivity than M&S within a specific range of low overallocation factors (ρ), a common scenario in FaaS. This suggests a significant advantage for SOGC in memory-constrained FaaS environments. While other GCs may achieve better performance in very specific scenarios—such as when memory is over-provisioned or when SOGC has a poor implementation—these are not the typical use cases for serverless applications. The COPY algorithm shows zero productivity when the overallocation factor is lower than 2, making it unsuitable for FaaS scenarios. Furthermore, our model indicates that all GCs experience a similar reduction in productivity with increasing allocation rates, highlighting the importance of understanding the algorithms' behavior under various workloads. The results from the SOGC setup scenario underscore the critical importance of efficient setup and initialization processes for the performance of our proposed algorithm, as a poorly optimized implementation can make SOGC a detrimental choice.

To further validate and extend this research, we propose future work in two main cate-

gories: improvements to the model and empirical evaluations. In terms of model improvements, we should:

- **Refine Model Parameters with Micro-benchmarks:** The parameters used in the analytical model should be refined by developing and executing micro-benchmarks that empirically measure the cost of individual GC operations. This would help to align the model with actual performance characteristics and increase its validity, ensuring our parameters are grounded in more realistic measurements.

- **Extend Model to Concurrent and Parallel Garbage Collection:** The model should be extended to support concurrent garbage collection algorithms and to address potential resource contention scenarios. This would enable the evaluation of different implementations and help in analyzing how concurrency can impact our findings, allowing for a better understanding of the algorithms' behavior in complex scenarios.

- **Analyze Model Sensitivity with Parameter Variations:** The model should be further analyzed by studying how it reacts to different parameter combinations. This can help us in understanding its limits and benefits, and should allow us to better study its relation to real-world scenarios, and also help us validate our assumptions.

To evaluate our model, we should:

- **Design, implement and evaluate a SOGC prototype:** Implement and evaluate the SOGC algorithm in a relevant FaaS environment. This would allow us to directly benchmark SOGC performance with specific FaaS workloads. By comparing the results with the predictions from our analytical model, we would accurately assess the model's validity and also understand the practical limitations of our approach, which uses a simplistic assembly mapping that ignores caching, reordering, various microarchitectural effects, and uses a flat memory model.

- **Refine Model Fidelity through Simulations:** Another approach would be to develop a simulation environment that allows for exploration of different mutator behaviors and memory usage patterns, as well as enabling analysis of the garbage collector performance under varying FaaS workloads. This would allow us to better understand the limitations of our approach and to create models that better capture the dynamics of

realistic FaaS environments. For example, in our analyzes, we considered that the mutator activity is constant (modelled by our λ parameter). If fact, it should be variable across the time.

This combination of future research and potential threats to validity aims to address the limitations of our model, to enhance it with more complex factors, and to further explore the potential of the proposed SOGC algorithm. We believe that this work provides valuable insight into the design and evaluation of garbage collection algorithms for FaaS environments. Further research is encouraged in this area to validate and expand our findings, focusing on more realistic scenarios and on the trade-offs between the different algorithms, with a particular focus on the optimization of the implementation of SOGC.

References

[1] Andrew W. Appel. “Garbage Collection can be Faster than Stack Allocation”. In: *Inf. Process. Lett.* 25.4 (1987), pp. 275–279. DOI: 10.1016/0020-0190(87)90175-X. URL: https://doi.org/10.1016/0020-0190(87)90175-X.

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. “A unified theory of garbage collection”. In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*. Ed. by John M. Vlissides and Douglas C. Schmidt. ACM, 2004, pp. 50–68. DOI: 10.1145/1028976.1028982. URL: https://doi.org/10.1145/1028976.1028982.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. “POPL 2003: A real-time garbage collector with low overhead and consistent utilization”. In: *ACM SIGPLAN Notices* 48.4S (2013), pp. 58–71. DOI: 10.1145/2502508.2502523. URL: https://doi.org/10.1145/2502508.2502523.

[4] Stephen M. Blackburn and Kathryn S. McKinley. “Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 22–32. DOI: 10.1145/1375581.1375586. URL: https://doi.org/10.1145/1375581.1375586.

[5] Stephen M. Blackburn et al. “Wake up and Smell the Coffee: Evaluation Methodology for the 21st Century”. In: *Commun. ACM* 51.8 (Aug. 2008), pp. 83–89. ISSN: 0001-0782. DOI: 10.1145/1378704.1378723. URL: https://doi.org/10.1145/1378704.1378723.

[6] João Carreira et al. “Cirrus: a Serverless Framework for End-to-end ML Workflows”. In: *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 13–24. DOI: 10.1145/3357223.3362711. URL: https://doi.org/10.1145/3357223.3362711.

[7] Perry Cheng and Guy E. Blelloch. “A Parallel, Real-Time Garbage Collector”. In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by Michael Burke and Mary Lou Soffa. ACM, 2001, pp. 125–136. DOI: 10.1145/378795.378823. URL: https://doi.org/10.1145/378795.378823.

[8] Marcin Copik et al. “SeBS: a serverless benchmark suite for function-as-a-service computing”. In: *Middleware ’21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*. Ed. by Kaiwen Zhang et al. ACM, 2021, pp. 64–78. DOI: 10.1145/3464298.3476133. URL: https://doi.org/10.1145/3464298.3476133.

[9] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Commun. ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794. URL: https://doi.org/10.1145/2408776.2408794.

[10] David Detlefs et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*. Ed. by David F. Bacon and Amer Diwan. ACM, 2004, pp. 37–48. DOI: 10.1145/1029873.1029879. URL: https://doi.org/10.1145/1029873.1029879.

[11] Edsger W. Dijkstra et al. “On-the-Fly Garbage Collection: An Exercise in Cooperation”. In: *Commun. ACM* 21.11 (1978), pp. 966–975. DOI: 10.1145/359642.359655. URL: https://doi.org/10.1145/359642.359655.

[12] Nafise Eskandani and Guido Salvaneschi. “The uphill journey of FaaS in the open-source community”. In: *Journal of Systems and Software* 198 (2023), p. 111589. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111589.

URL: https://www.sciencedirect.com/science/article/pii/S0164121222002655.

[13] Lang Feng et al. “Exploring Serverless Computing for Neural Network Training”. In: *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE Computer Society, 2018, pp. 334–341. DOI: 10.1109/CLOUD.2018.00049. URL: https://doi.org/10.1109/CLOUD.2018.00049.

[14] Daniel Fireman et al. “Improving Tail Latency of Stateful Cloud Services via GC Control and Load Shedding”. In: *2018 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2018, Nicosia, Cyprus, December 10-13, 2018*. IEEE Computer Society, 2018, pp. 121–128. DOI: 10.1109/CLOUDCOM2018.2018.00034. URL: https://doi.org/10.1109/CloudCom2018.2018.00034.

[15] Daniel Fireman et al. “Prebaking runtime environments to improve the FaaS cold start latency”. In: *Future Gener. Comput. Syst.* 155 (2024), pp. 287–299. DOI: 10.1016/J.FUTURE.2024.01.019. URL: https://doi.org/10.1016/j.future.2024.01.019.

[16] Daniel Lacet de Faria Fireman, Raquel Vigolvino Lopes, and João Arthur Brunet Monteiro. “Improving tail latency of interactive cloud microservices through management of background tasks”. In: *SISTEMOTECA - Sistema de Bibliotecas da UFCG*. 2021.

[17] Christine H. Flood et al. “Parallel Garbage Collection for Shared Memory Multiprocessors”. In: *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. Ed. by Saul Wold. USENIX, 2001. URL: http://www.usenix.org/publications/library/proceedings/jvm01/full%5C_papers/flood/flood.pdf.

[18] Christine H. Flood et al. “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. PPPJ '16 (2016). DOI: 10.1145/2972206.2972210. URL: https://doi.org/10.1145/2972206.2972210.

[19] Jürgen Heymann. "A comprehensive analytical model for garbage collection algorithms". In: *ACM SIGPLAN Notices* 26.8 (1991), pp. 50–59. DOI: 10.1145/122598.122602. URL: https://doi.org/10.1145/122598.122602.

[20] Pedro García López et al. "ServerMix: Tradeoffs and Challenges of Serverless Data Analytics". In: *CoRR* abs/1907.11465 (2019). arXiv: 1907.11465. URL: http://arxiv.org/abs/1907.11465.

[21] Horácio Martins, Filipe Araújo, and Paulo Rupino da Cunha. "Benchmarking Serverless Computing Platforms". In: *J. Grid Comput.* 18.4 (2020), pp. 691–709. DOI: 10.1007/s10723-020-09523-1. URL: https://doi.org/10.1007/s10723-020-09523-1.

[22] David Quaresma, Daniel Fireman, and Thiago Emmanuel Pereira. "Controlling Garbage Collection and Request Admission to Improve Performance of FaaS Applications". In: *32nd IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2020, Porto, Portugal, September 9-11, 2020*. IEEE, 2020, pp. 175–182. DOI: 10.1109/SBAC-PAD49847.2020.00033. URL: https://doi.org/10.1109/SBAC-PAD49847.2020.00033.

[23] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. "Prebaking Functions to Warm the Serverless Cold Start". In: *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. Ed. by Dilma Da Silva and Rüdiger Kapitza. ACM, 2020, pp. 1–13. DOI: 10.1145/3423211.3425682. URL: https://doi.org/10.1145/3423211.3425682.

[24] Nikhila Somu et al. "PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications". In: *2020 International Conference on COMmunication Systems & NETworkS, COMSNETS 2020, Bengaluru, India, January 7-11, 2020*. IEEE, 2020, pp. 144–151. DOI: 10.1109/COMSNETS48256.2020.9027346. URL: https://doi.org/10.1109/COMSNETS48256.2020.9027346.

[25] Mads Tofte and Jean-Pierre Talpin. “Region-based Memory Management”. In: *Inf. Comput.* 132.2 (1997), pp. 109–176. DOI: 10.1006/INCO.1996.2613. URL: https://doi.org/10.1006/inco.1996.2613.

[26] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. “Low-latency, high-throughput garbage collection”. In: *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 76–91. DOI: 10.1145/3519939.3523440. URL: https://doi.org/10.1145/3519939.3523440.

Appendix A

Gargage collector operations to Assembly

A.1 Mark-and-Sweep

A.1.1 Mark

Source Code A.1: Assembly code for Marking operation

```
1    MOV rdi, object_address
2    MOV rbx, bitmask
3    MOV [rdi], rbx
```

This code represents the marking operation by loading the memory address of the object and then using the bitmask to activate a specific bit in memory. This bit will represent if this object is a live object or not.

A.1.2 Scan

Source Code A.2: Assembly code for Scanning operation

```
1    MOV rax, [rdi]
2    CMP rax, heap_start
3    JL next_node
4    CMP rax, heap_end
5    JG next_node
6        next_node:
```

This code demonstrates how a GC checks if a memory address is a pointer to a memory location within the heap. The code loads the value of a memory node in the register rax, and then checks if that address is between the start and end of the heap. This code does not show how the GC actually gets to the next node in the heap, and how it will keep processing the heap nodes, as this snippet aims to model a single iteration in the scanning process.

A.1.3 Allocation

Source Code A.3: Assembly code for Allocation operation

```
1    MOV rbx, 1
2    MOV [rdi], rbx
```

This code shows how the GC will perform the allocation of a new node, by simply marking a memory region as occupied, with the value 1.

A.1.4 Reclamation

Source Code A.4: Assembly code for Reclamation operation

```
1    MOV rbx, 0
2    MOV [rdi], rbx
```

This code demonstrates how the GC reclaims the memory of a node, by simply setting a specific location in memory to the value of zero.

A.2 Copy-Live-Objects

A.2.1 Copying

Source Code A.5: Assembly code for Copying operation

```
1    MOV rax, [rdi]
2    MOV [rbx], rax
```

This code represents the copy operation by loading the value from a specific memory location, and then storing this value in the other memory location.

A.2.2 Allocation

Source Code A.6: Assembly code for Allocation operation

```
1    MOV rbx, 1
2    MOV [rdi], rbx
```

This code shows how the GC will perform the allocation of a new node in tospace, by simply marking a memory region as occupied, with the value 1.

A.3 Serverless-Optimized Garbage Collector

A.3.1 Persistent Memory Setup

Source Code A.7: Assembly code for Persistent Memory Setup operation

```
1    MOV rbx, value1
2    MOV [rdi], rbx
3    MOV rbx, value2
4    MOV [rdi+offset1], rbx
```

This code simulates the process of setting up the persistent memory of a handler, which involves loading values to a register and then storing them in a given memory address.

A.3.2 Handler Allocation

Source Code A.8: Assembly code for Handler Allocation operation

```
1    MOV rbx, 1
2    MOV [rdi], rbx
```

This code demonstrates how a GC allocates a new node for the handler, by simply marking a given memory location as occupied by setting a value in memory.

A.3.3 Handler Reclamation

Source Code A.9: Assembly code for Handler Reclamation operation

```
1       MOV rbx, 0
2       MOV [rdi], rbx
```

This code demonstrates how the GC reclaims the memory of a node from the handler, by simply setting a specific location in memory to the value of zero.