Multithreading Support in GarCoSim

by

Andrii Kuch

B.Sc. in Computer Science, Sevastopol State Technical University, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, Ph.D., Computer Science
	Gerhard Dueck, Ph.D., Computer Science
Examining Board:	Patricia A. Evans, Ph.D., Computer Science (chair)
	Huajie Zhang, Ph.D., Computer Science
	Arash Habibi Lashkari, Ph.D., Computer Science
	Richard Tervo, Ph.D., Electrical and Computer Engineering

This thesis is accepted dy the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2018

Abstract

Garbage Collection Simulator (GarCoSim) is a framework that has been created to ease the analysis of existing memory management techniques, as well as the prototyping of new memory management techniques. As a part of the framework, the Trace File Simulation tool has been created to prototype and test new Memory Allocation and Garbage Collection (GC) policies. This thesis focuses on the enhancement of the functionality of the simulator, namely on enabling the simulator to run multiple threads. Such a simulation would represent processes in memory more realistically, as well as allowing prototyping and testing thread-aware garbage collection methods. This work describes the changes that have been made to the simulator and its trace files to make such a parallel execution possible.

Contents

List of Figures

Chapter 1

Introduction

Modern programming languages like Java rely on automatic memory management (AMM). It is a technique in which a system automatically manages the allocation and de-allocation of memory. A Java Virtual Machine (JVM) is an example of such a system: while running, it automatically frees memory occupied by objects, that cannot be reached anymore. This automated memory freeing is called Garbage Collection (GC). This simplifies the process of software development, as now the developer does not need to delete objects manually. AMM can eliminate common problems such as forgetting to free memory allocated to an object and causing a memory leak, or attempting to access memory for an object that has already been freed.

Advantages of automated GC come at a cost of possible long pauses and uneven performance and also can impact application responsiveness. Different

GC algorithms provide different approaches to when and how much GC needs to be performed, as well as which section of memory needs to be localized for a GC. As GC techniques are being developed and new algorithms are being introduced, it is necessary to prototype and to test new approaches. However, the JVM is a large-sized project, implementing a new feature is a time consuming task, while it is desired to prototype a solution fast. GarCoSim is a framework that allows developers to simulate memory management processes in the JVM, and it is a convenient tool for studying existing and prototyping new GC techniques. [22] One of the approaches in GC that is aimed to fight uneven performance during the collection is thread-aware GC. The reason for long pauses and performance drop during GC is "stop-the-world" situation. During the GC, the part of the tree of objects that is to be collected should remain unchanged. Clearly, if the entire heap is being collected, this means that all threads (except the collector) should halt until the collection is finished. Thread-aware GC techniques attempt to memorize which threads have allocated which objects and to layout the objects on the heap in such a way, that GC for certain threads could be performed without pausing others.

This work is focusing on the question of enabling GarCoSim to model thread-aware GC by adding multithreading functionality to the simulator. The reader will have a close look into the structure of GarCoSim simulator, then its current implementation and its limitation will be discussed. Later in this work, a design and implementation of a multi-threaded version of the

simulator is presented, experimental results are discussed.

Chapter 2

Background

This chapter will give an overview of the background and technologies that are used in this thesis.

2.1 Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract computing machine [18]. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. A JVM translates the Java programming instructions into instructions and commands that run on the local operating system. This way, Java programs achieve platform independence [18]. Once a Java virtual machine has been implemented for a given platform, any Java program, which, after compilation, is called bytecode, can run on that platform. A Java virtual machine can either interpret the bytecode one

instruction at a time (mapping it to a real processor instruction) or the byte-code can be compiled further for the real processor using what is called a just-in-time compiler.

Garbage collection is an integral part of the JVM as it collects unused Java heap memory so that the application can continue allocating new objects. The effectiveness and performance of the GC play an important role in application performance and determinism [6].

2.2 Garbage Collection in JVM

In many programming languages (e.g., C, and C++) dynamically allocated memory must not only be tracked by the programmer, but must also be freed when it is no longer needed. Tracking and freeing dynamically allocated memory is a time-consuming task performed by programmers who are error-prone. In Java and other languages (e.g., Lisp, Smalltalk, Python) the runtime system keeps track of memory or objects that have been dynamically allocated and periodically frees the memory that is no longer being used, i.e., it automatically performs garbage collection.

In several Java implementations, garbage collection is performed synchronously. In other words, the executing program is suspended for a period of time while garbage collection is performed. Alternatively, some approaches to garbage

collection attempt to simultaneously execute the garbage collector code and the main application by using a separate thread of control for garbage collection. However, the suspension of the main application can cause serious problems for users or other programs attempting to interact with the application [7].

There are two reasons why GC has a significant effect on overall execution time. There is a direct impact, which is the time spent for garbage collection itself. In their work Singer, Brown et al.[27] show that the mean proportion of execution time spent (direct impact) in GC is 12.2%. There is also indirect impact, which is caused by the manner in which the GC algorithm rearranges heap-allocated data after collection. This can affect subsequent program execution time due to significant changes in the spatial locality of data [14].

2.3 GarCoSim

If a new GC algorithm or a modified approach of an excising GC technique needs to be evaluated, it is often to be implemented in a real JVM. Subsequent evaluation is performed using various benchmarks. In their work Nasartschuk, Dombrowski et al.[22] introduce an approach of creating an evaluation framework for garbage collection techniques that does not rely on a specific virtual machine. The idea of their work is to provide a memory management simulator, which replaces a JVM with a simulation of its

memory management capabilities. The application is replaced by a trace file, which is extracted from an existing virtual machine. In addition, the framework is developed in an extendable, open source form, so researchers can evaluate and compare new techniques to the results of others in an isolated environment [22].

The main idea of GarCoSim is to be as simple as possible. Each virtual machine offers a number of optimization methods on a variety of levels, such as garbage collection, allocation, just-in-time compilation and object modelling. With such a number of parameters comparing different techniques can be complicated and can take too much time. GarCoSim, on the other hand, provides a simple and safe "sandbox" environment, where new or modified algorithms can be implemented independently of actual implementation of a specific virtual machine.

GarCoSim uses trace files captured during an actual run of an application using an instrumented (modified for this purposes) JVM. Trace files offer the flexibility of being able to develop a tailored simulator that focuses only on the basic MM operations and at the same time, be portable to be compiled and run on different platforms [22]. A trace file is a collection of basic memory management operations typically found in executing an application. Table 2.1 [22] provides a list of all instructions that can be found in a trace file.

Memory Management Operations	Notation	Usage
Allocation	a	T_i O_j S_k N_l C_j
Add a reference of an object to the rootset a thread	+	T_i O_j
Store/write a primitive field into an object	s	T_i O_j $I_x(/F_m)$ S_n V_o
Store/write a static primitive field into a class object	s	T_i C_j $I_x(/F_m)$ S_n V_o
Store/write an object reference field into an object	w	T_i P_j #k O_l F_m S_n V_o
Store/write a static object reference field into a class object	c	T_i C_j $I_x(/F_m)$ O_l S_n V_o
Read a primitive or a reference field from an object	r	T_i O_j $I_x(/F_m)$ S_n V_o
Read a static primitive or reference filed from a class object	r	T_i C_j $I_x(/F_m)$ S_n V_o
Delete an object reference from the rootset of a thread	-	T_i O_j

T_i: i is a thread id (i >= 0)
O_j: j is an object id (j >= 1)
C_j: j is a class id (j >= 1)
S_k: k is the size of a object in bytes (k >0)
N_l: l is the number of reference slots in an object (l >= 0)
$I_x(/F_m)$: x(/m) is the index(/offset) of a filed in an object (x(/m) >= 0)
S_n: n is the size of a field in an object (n >= 8)
V_o: o is either 0 or 1 represents the field type either non-volatile or volatile
P_j: j is an object (Parent) id (j >= 1)
O_l: l is an object (Child) id (l >= 1)
#k: k is the slot number of a reference field in an object

Table 2.1: The trace file instructions

GarCoSim is a suitable environment for this thesis project, because it provides a relatively simple framework that allows developers and researchers to concentrate on garbage collection itself without relying on a specific virtual machine or worrying about optimization of irrelevant parameters.

2.4 Trace File Generator

In addition to the simulator, the GarCoSim framework also has a tool for generating trace files. This generator was designed to automatically generate different types of basic memory management operations and write into a synthetic trace files [3]. Generated trace files realistically mimic behaviour of a multi-threaded application that is allocating objects, accesses them for

writing and reading and eventually stops using them (the objects become garbage). Generated trace files follow the same specification as captured trace files (see Table 2.1). Moreover, generated trace files mimic such important properties of trace files as length of the trace file, number of operating threads, percentage of allocations, store and access operations, ratio of static field access, etc. The default values of these parameters are picked based on statistics real trace files provide, however all these parameters can be also reassigned manually.

The trace file generator is a convenient tool for this thesis project as it can provide trace files with correct inner structure, realistic behaviour and still, with desired characteristics such as certain length of the trace file or number of threads that are running.

Chapter 3

Application of Machine Learning Techniques to Garbage Collection in GarCoSim

The work that is described in this chapter is a preliminary investigation of how garbage collection can be improved. This work was completed in GarCoSim, a framework under development for garbage collection research [22]. This work was important to understand the inner organization of the GarCoSim framework, fundamentals of GC techniques and implementation of GC algorithms in GarCoSim.

3.1 Background

The work described in this chapter combines machine learning techniques in garbage collection in automatic memory management. This section gives a short introduction to both fields.

The Section 3.1.1 briefly describes how garbage collection works in a JVM and how the collection process can be manipulated. Section 3.1.2 gives a short introduction into the Reinforcement learning field, which is important to understand how these techniques can be potentially applied to garbage collection.

3.1.1 Balanced GC

The primary goal of the Balanced Collector is to amortize the cost of global garbage collection across many GC pauses, reducing the effect of whole-heap collection times. At the same time, each pause should attempt to perform a self-contained collection, returning free memory back to the application for immediate reuse. To achieve this, the Balanced Collector uses a dynamic approach to select heap areas to collect in order to maximize the return-on-investment of time and effort [26].

The Balanced Garbage Collection policy uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum

pause time on large heaps. The Java heap is split into potentially thousands of equal sized areas called regions. Each region can be collected independently, which allows the collector to focus only on the regions which offer the best return on investment.

Objects are allocated into a set of empty regions that are selected by the collector. This area is known as the eden space. When the eden space is full, the collector stops the application to perform a Partial Garbage Collection (PGC). The collection might also include regions other than the eden space, if the collector determines that these regions are worth collecting. When the collection is complete, the application threads can proceed, allocating from a new eden space, until this area is full. This process continues for the life of the application [1].

The set of regions that is included into PGC is called a collection set. It is used to collect garbage from regions with a high mortality rate and to collect and defragment regions outside of the eden set. It easy to see that by manipulating a collection set, overall performance of the Balanced Collector can be improved. As only part of the heap is selected for garbage collection, it is important to select a part that is big enough to return a sufficient amount of free memory to the allocator and still not too big, otherwise garbage collection will take too much time. It is also very important to select a proper part of the heap for garbage collection, the one that promises best return-

on-investment, meaning it has higher probability to return more memory.

The idea that an efficiently picked collection set will result in a better garbage collection is the key idea of this project. In this project we plan to investigate if it is possible to automatically find a way of constructing a collection set that is more promising from the perspective of returned memory.

3.1.2 Reinforcement Learning

Reinforcement learning is an area of machine learning inspired by behaviorist psychology. It is aimed at controlling a computer agent so that a target task is achieved in an unknown environment [28] so as to maximize some reward [25]. In other words, reinforcement learning is learning how to map situations to actions so as to maximize a numerical reward signal. The agent is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics – trial-and-error search and delayed reward – are the two most important distinguishing features of reinforcement learning [29].

Typical reinforcement learning problems consist of an acting agent, set of actions an agent can do, an environment agent is acting in (represented by a set of stages), and a reward associated with an action taken in a given state.

The agent does not know the reward before an action is taken and has to perform the action to learn the outcome. By trying different actions (often many times) in different stages the agent accumulates experience. The agent's goal is to maximize the reward by picking the best action. It is important to note that a clever strategy is not only to try to maximize the immediate reward, but also plan ahead and pick an action that not only maximizes the current reward, but also allows to take better steps in future.

The idea of this project is that the reinforcement learning approach is applicable to the garbage collection problem because reinforcement learning has following properties:

- **The agent is concerned about delayed (cumulative) rather than immediate reward.** The collector is supposed to maximize efficiency not only of current collection but of all collections in a long run.

- **A model of the environment is known, but an analytical solution is not available.** At every point of execution it is possible to perform garbage collection with arbitrary parameters and see its result, but it is feasible to pre-calculate all possible collection runs with all possible parameters in all possible sequences. Neither can we see how the running application is going to allocate memory in future and based on that predict the best sequence of actions. In other words, we

can see results of a single step and learn from it but we cannot analyze all possible outcomes in future and derive the best sequence of actions from it.

- **Interaction with the environment happens in discrete time steps.** At every step the agent should pick an action in a given state, perform it and face the consequences (obtain reward). This is happening during garbage collection: once garbage collection is required, the collector picks a collection set, performs the collection and obtains freed memory as a reward.

- **The only way to collect information about the environment is by interacting with it.** Meaning there is no way to know in advance how much memory the collection will return. The only way to obtain this information is actually to perform the collection with given parameters.

- **What is done cannot be undone.** The agent cannot cancel its actions and their effect. In other words, if a poor collection was performed there is no way to undo it and try again. Instead, the agent will accumulate experience, in this case a negative one.

In this chapter we introduced the main components of the project. The chapter starts with a description of related work. Later, a particular technique is discussed – Balanced Garbage Collection – that is going to be used in the

project. Finally, a short introduction into reinforcement learning is presented and its applicability of this particular approach is explained.

3.2 Related Work in Machine Learning

Very few references can be found where researchers try to apply reinforcement learning techniques to the garbage collection problem.

In her work [4] Andreasson investigates how machine learning methods can enhance garbage collection in a JVM. The author analyzes the suitability of other prominent machine learning methods, such as neural networks, decision trees and genetic algorithms, to the problem of GC and comes to a conclusion that reinforcement learning is the most suitable. Andreasson presented a solution where the optimal time to trigger a GC is learned based on observations like the heap fragmentation factor, the speed at which the running program allocates memory and the average size of new allocated objects. The presented adaptive solution demonstrates performance similar to human designed heuristics, however it was not able to surpass them.

In their work Kang et al.[17] address the problem of long-tail latency in NAND flash memory-based storage systems and propose a reinforcement learning-assisted garbage collection technique, which learns the storage access behavior online and determines the number of GC operations to exploit the

idle time. Although the proposed solution is very interesting and was able to outperform state-of-the-art techniques, it addresses a very specific hardware issue of SSD and obtained techniques cannot be directly applied to systems, that use other types of memory.

No other references to the application of reinforcement learning to the garbage collection problem were found. In both presented cases the authors refer to a difficulty of representing the problem in a form that accepts application of reinforcement learning, especially pointing out the trade-off between number of valuables (state features) and the increased time required for learning due to an enlarged state space.

3.3 Design

This section specifies the problem the project is aimed to solve, create an approach on how to solve the problem and how to create an evaluation strategy.

The problem specification can be found in Subsection 3.3.1. An overview of the approach is located in Subsection 3.3.2. Subsection 3.3.3 describes how the application of the intended design is going to be implemented and how it can be evaluated.

3.3.1 Problem Specification

Different applications naturally have different memory usage patterns. A computationally intensive number crunching workload will not use the Java heap in the same way as a highly transactional customer-facing interface [6].

The hypothesis "Objects die young" states that, in typical applications, most allocated objects have a short life span and can be collected shortly after they have been allocated [6]. Generational collectors, including Balanced [26], leverage this by designating areas in the heap, typically referred to as new space, that are used for object allocation and are collected by specialized collectors. This seeks the best return-on-investment of time and effort to reduce total pause times relative to whole heap collection. Eventually, as objects survive long enough to leave the new space, the remainder of the heap, referred to as old space, fills up and must be collected with a whole heap collector [26].

Although, the general idea that younger objects have higher mortality rate is clear, the question remains unanswered if different applications have exactly the same pattern of objects' mortality rate depending on age. In other words, is it known in advance that for any an application object of a given age has a corresponding probability of being dead, or because of using heap differently, different applications will benefit from different parameters of garbage collection.

As described in Subsection 3.1.1, this project deals with the Balanced Garbage Collector. Parameters of garbage collection in this context mean how the collection set was picked.

The problem the project is aiming to solve can be specified as follows: to investigate if a garbage collector can automatically adjust its parameters during runtime in such a way, that the picked collection set will allow garbage collection to be more efficient for a given application. It is expected that the project will either propose a solution that leverages found differences between applications and learns from them, or an explanation why it is problematic or not possible.

3.3.2 Approach Overview

It is well-known that different garbage collectors work best for different programs. A key point is that it is not at all obvious how to determine which GC algorithm works well with each program and heap size [27]. On the other hand, even if an optimal collector and heap size are known, there are different ways the collector can be fine-tuned for a particular application. It would be better, however, to derive this kind of relationship automatically using machine learning techniques.

As described above, this project deals with an application of reinforcement

learning techniques to the Balanced Garbage Collector. It is planned to manipulate the collection set for every garbage collection in order to achieve a more efficient collection. To do so the collector should, at the beginning of every collection, pick an action that changes how the collection set is going to be created.

Let us assume at the beginning of every garbage collection, the set is created probabilistically based on the age of regions. According to the specification of the Balanced Collector, the eden space (all regions with age 0) are always included (probability is 1) [26]. Older regions are included with lower probability. The probability continues to decrease to some point and stays linear after, as it is desirable that regions of all ages can potentially be included into the collection set (their probability never drops to 0). Then the probability function will look as on Figure 3.1

The shape of the function can be described by two parameters: a – the slope, and b – constant probability for older regions. By varying the parameters a and b, the probability function can be reshaped. The required number of regions can be sampled using the given function.

Parameters a and b are continuous by nature, however they can be discretized. Then discrete variations of a and b will create a set of variations. In the example on Figure 3.2 both parameters have five different values. this

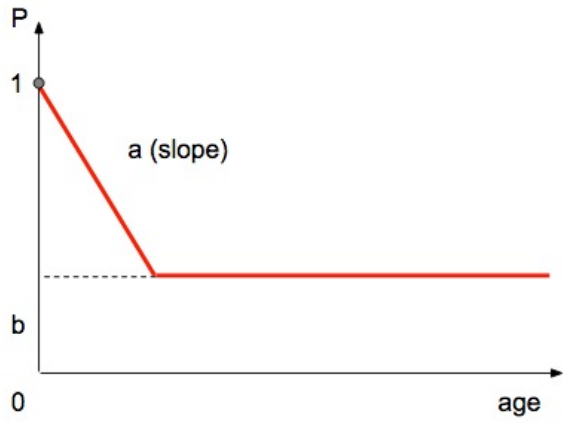

Figure 3.1: Probability function showing dependency of being selected to collection set based on age

provides 25 different variations of these two parameters. They create a space where each state corresponds to a certain combination, adjacent states differ from each other by one step change of one of parameters.

The space depicted in Figure 3.2 is the environment, the world, for the agent (Collector) to explore. Starting at a random position in the world, the agent starts with certain shape of the probability function. Actions that are available are moving one step to adjacent state, meaning increasing or decreasing a or b by one step.

After trying different states (i.e., shapes of the probability function) the collector during each collection obtains a reward – how much memory was freed. For this project we consider the number of freed objects to be the reward, i.e.,

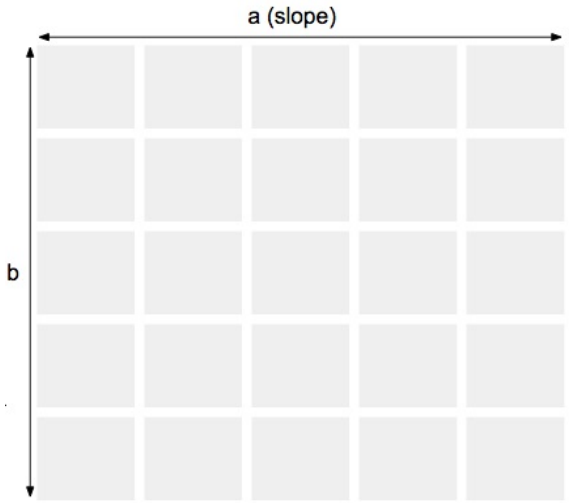

Figure 3.2: World to explore created by discrete variations of parameters a and b

the size of the objects is ignored. This decision was taken for convenience: it is easier to count the reward in thousands (freed objects), rather than in tens or hundreds of millions (memory in bytes).

It is assumed that over time the collector will learn moving from state to state and will be able to generalize the knowledge, that there are states that are more promising than others, and the collector will start to spend more collections in this promising states or in the adjacent states. In other words, over time the collector should determine "favorite" probability configurations that give a better reward (returns more objects).

3.3.3 Implementation Plan and Evaluation Concept

The intended design has to be implemented on top of a traditional Balanced Garbage Collector. As a framework for implementation and evaluation of the project, GarCoSim was chosen. Thus, steps of the project are the following:

1. Implementation of traditional Balanced Collector in GarCoSim;
2. Implementation of described machine learning mechanism;
3. Comparison of performance of both implementations using same files.

3.4 Implementation

This section describes how the proposed design was implemented. Section 3.4.1 describes how traditional Balanced garbage collection was implemented in GarCoSim. Section 3.4.2 describes how a reinforcement learning mechanism was implemented on top of traditional Balanced GC.

3.4.1 Balanced GC in GarCoSim

General information about Balanced GC was gathered from series of articles by IBM [6], [26] and other literature [15]. There are main rules for Balanced GC that describe the heap layout and give some information about collection sets. These rules used for implementation of Balanced GC in GarCoSim are:

- The heap is split into regions of same size;

- the number of regions must be less than 2048;
- The size of each region is the smallest power of two (512KB, 1MB, 2MB, 4MB, etc.) that results in less than 2048 regions;
- 25% of all regions are eden space. New objects will be allocated there only.
- Only a subset of regions is picked as a collection set. Garbage collection is performed only over a part of the heap;
- Eden regions are always included in the collection set.

It is important to note that collection sets are used only in Partial Garbage Collection (PGC). There is also Global Garbage Collection (GGC) which is performed using mark-sweep-compact. However, the overall idea of Balanced GC is to avoid global collections by all means. Ideally, GGC should never happen [26].

For PGCs to be able to accurately discover all live objects, the collector must scan all object roots (for example, thread stacks, permanent class loaders, JNI references). Additionally, all references to objects within the collection set from objects external to the collection set must be discovered. This could be accomplished by scanning all objects in all regions not included in the collection set, but this would be very inefficient. Instead, references between objects in different regions are tracked and recorded in a data structure known

as the remembered set. The remembered set is used to track all in-bound references on a per-region basis [26].

Remembered sets (remsets) are very important as they allow the collector to discover all live data, even if references from rootset to current object are going through regions not included in collection set and thus cannot be discovered from the rootset directly. Because of that remsets should be traversed just like rootsets.

To discover all live data and in order for all pointers to be updated properly (in case pointed objects were moved), all remsets (belonging or not belonging to collection set) should be traversed, because objects in the collection set can be referenced from objects outside the set, and the other way around. Because of this, several traversals are needed: over the rootset, over the remsets in collection set, and in the remsets outside the the collection set.

Another important concept that is used in the implementation is the forwarding pointer. It is used when a certain object was already moved during garbage collection. It points from the old instance of the object to the new one, so if the same object is reached again through other objects, all references can be updated correspondingly.

According to everything described above, the Balanced GC algorithm was

implemented as a sequence of following steps:

1. Building of the collection set;
2. Traversing the rootset and copying objects;
3. Traversing the remsets that belong to the collection set;
4. Traversing the remsets that do not belong to the collection set. This phase is needed to update (if objects were moved during the collection) or delete (if the objects are garbage) pointers from outside of the collection set.

3.4.2 Reinforcement Learning Algorithm

For the implementation of a reinforcement learning algorithm a Q-learning approach was chosen. Q-learning can be used to find an optimal action-selection policy for any given decision process [31]. It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state.

In Q-learning problem model consists of:

- Representation of environmental states s;
- Set of possible actions a;

The agent can move from state to state. Executing a particular action in a specific state provides the agent with a reward. Value Q is referred to as state action value. Before learning has started, Q should return an arbitrarily fixed value, 0 in our case. Later, during the run, the agent selects an action, and observes a reward and a new state that may depend on both the previous state and the selected action, Q is updated. Depending on an obtained reward certain actions become more desirable than others. In the end there is a set of preferable actions for every state – policy – after a sufficient number of trials the policy is expected to converge to an optimal one.

3.4.2.1 Q-value Update Rule

To choose an action, the algorithm simply picks a higher expected reward. If there are several actions that promise the same reward (for example at the very beginning of learning where all initial Q values are 0), a random action is chosen. It is easy to see that such an approach considers only the immediate (one step ahead) reward. It is desirable, however, to include a look-ahead value, so that the chosen action does not only provide immediate reward but also cares about rewards in future. This is done as follows. When the current Q value is being updated for the state-action combination that was

just experienced, the algorithm will also search over all Q values reachable directly from reached state, find the maximum and incorporate it into the Q value update for state-action combination that was just experienced. In other words, the Q value is being updated not only by the immediate reward, but also by the best expectation reachable from the new state. There are different ways to incorporate future expectations into Q value updates. Usually only a certain number of forward steps are considered, and the algorithm is looking only that far into future. It is common to introduce a discount factor γ for future rewards, making future rewards less valuable than immediate reward. In this project it was chosen to look only one step ahead, because the changing nature of the environment makes looking further useless. In this project the future reward (the reward of next upcoming action) is not discounted and considered as important as immediate reward. In other words, in this particular implementation the value of γ is set to 1.

Finally, an update of the Q value is multiplied by a learning rate α. A learning rate is a very common practise in all machine learning approaches [29]. It is introduced to make changes to values not so significant, to slow down or to smoothing changes over time. In many machine learning approaches it helps to avoid solution osculations. The default value of α is often chosen between 0.1 and 0.9 depending on how carefully or aggressively the algorithms is tuned to move towards the solution. This value is often picked experimentally and the motivation behind the choice is to balance between

the convergence speed and the risk of an unstable learning process. in this implementation the default value of α was set to 0.2.

To summarize all rules of updating of Q value, we obtain a following general rule of Q value update:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Figure 3.3: Q value update rule

3.4.2.2 Exploration vs. Exploitation

A common problem in reinforcement learning is the exploration versus exploitation dilemma [29]. The problem can be described as follows: when an agent has partial information about possible rewards (knows only some rewards for only some actions) will it be optimal to use already obtained knowledge for a guaranteed reward or to take actions with unknown outcome in a hope for a better solution. In other words, should the agent be cautious (and go only for known actions) or curious (inspired to explore unknown part of environment).

A common solution for this problem in Q-learning (and reinforcement learning in general) is an introduction of an exploration factor ϵ. In many implementations ϵ is simply a probability of random action being taken instead of

the best known one. There are many common practices to handle ϵ value. Often ϵ is designed to decrease over time to discourage the agent from exploring new possibilities, to settle down and simply reuse obtained experience. However, due to rapid changes in the explored environment in this project it was decided to make ϵ value static.

3.5 Results Evaluation

This section describes how implementation described in Section 3.4 can be evaluated and how it answered questions asked in Section 3.3.1.

3.5.1 Evaluation Approach

As it was described in Section 3.4, to answer the questions asked in this project several steps were taken:

- Balanced GC was implemented in GarCoSim simulator;
- The collection set creation mechanism was altered in this Balanced GC implementation in order to introduce a machine learning mechanism based on reinforcement learning.

The next step is to evaluate this implementation and to answer the question if application of reinforcement learning was actually able to improve perfor-

mance of garbage collector.

However, this evaluation can be problematic because of following reasons:

- Comparing two implementations of Garbage Collectors in GarCoSim is questionable, because it cannot be assumed that traditional Balanced GC performs optimally. On the other hand, the algorithm can be also compared to itself with ϵ equals 1.0, i.e., the situation when all taken actions are random.

- Garbage collection in the current approach is highly stochastic and because of this algorithm's performance is hard to measure. On the other hand, we are more interested if the algorithm converges to a particular solution, i.e finds one configuration better than others.

- Finally, even if a solution can be found, presented approach is promising only for the tested application. For different applications different solutions will be optimal.

Taking into consideration the problems mentioned above, there are questions that are to be answered:

- Does the algorithm show reasonable behaviour that is better than random behaviour?

- If yes, does the algorithm show a tendency to converge to a particular solution?

- If yes, does the solution differ for different applications (trace files)?

It is important to note that answers to these questions should be not only given but also explained.

3.5.2 Observations Analysis and Discussion

Testing runs of the implemented solution showed that the algorithm does try to explore different states, receives reward and updates Q-values of given state-action combination. However following problems were noticed:

- Apparently, the implemented algorithm needs numerous runs to accumulate enough experience. That means numerous garbage collections (tens to even hundreds) should happen. In GarCoSim the application is replaced by a trace file which is extracted from an existing virtual machine [22]. This naturally limits the number of possible garbage collections as running is limited by length of the trace file. In other words, the length of available trace files is not enough for the algorithm to perform sufficient number of learning steps.

- In the beginning the algorithm is too tempted to reuse first-obtained rewards and because of that is not willing to explore unknown space much. This can be compensated by increasing ϵ values in the beginning (and reducing later), but this only increases the number of steps needed for learning, which only makes the previously described problem worse.

Because of these two problems mentioned above, results about gathered Q-values are limited. However, it is possible to notice that the obtained reward values (number of freed object) are very similar in most cases. This leads to an assumption that even if it was possible to make enough runs, probably rewards landscape over all possible states would be too flat for the algorithm to figure out direction of optimum.

To summarize, the presented approach was unable to demonstrate benefit for the Balanced Garbage Collection as designed. As possible reasons for that we would specify the following:

- Picked implementation environment is not suitable for the taken approach. Implementation in a real JVM, where an application can run for a longer time and can have numerous garbage collections, could be more suitable.
- Suggested design introduces too many stochastic values that add noise to results and slows down learning (or makes it impossible). A more deterministic model of creating of the collection set can potentially show better performance.

3.6 Conclusion and Recommendations

This work was aimed to answer the question if reinforcement learning can be potentially applied to the problem of garbage collection in the Java Virtual

Machine.

To answer this question a solution was designed that uses a modified version of the Balanced Garbage Collector, where collection sets are generated based on reinforcement learning using a probabilistic model. The proposed solution was implemented in GarCoSim environment.

Testing runs did not show beneficial results. Performance of the modified collector was comparable to traditional Balanced GC. Possible reasons were discussed in Section 3.5.2.

Recommended next actions may include trying alternative machine learning solutions, that are more deterministic than the proposed solution. For example it can be an application of a neural network, that classifies a region as desirable or not desirable in the collection set based on a set of features like age, fragmentation, expected mortality rate.

Another important conclusion drawn during research was that as machine learning techniques often require numerous learning steps, an actual virtual machine, where a continuous run of an application over a long period of time is possible and can be a more suitable environment for an implementation of these techniques.

Chapter 4

GarCoSim Parallelization Problem Formulation and Solution Design

This chapter discusses in detail the problem of parallelizing GarCoSim for accurate multi-threaded garbage collection simulations:

- What is the current state of GarCoSim and why it is not sufficient;
- How GarCoSim trace files work and why it is challenging to split them thread-wise;
- How the parallelization can be approached.

Later in this chapter a solution design is described in detail. Implementation of the solution is described in the next chapter.

4.1 Parallelization Problem Statement

The main purpose of GarCoSim is to provide a lightweight GC development framework that is easier to to understand and maintain than a full-scale JVM. At the same time, there is an increasing interest in parallel GC algorithms, where the stop-the-world phase is avoided as only some threads are stopped for collection of data that is associated with them. Naturally, this means that GarCoSim should be able to operate physically separate threads in order to model such parallel GC algorithms. However at the moment, simulation in GarCoSim is strictly sequential: every memory operation is executed in the order they were written to a trace file; information about threads that performed these operation is not used during actual simulation. The key problem this thesis addresses is to enable GarCoSim to operate actual separate threads in order to make simulation more realistic. In this section the reader will be presented a closer look at the inner structure of trace files and investigate how thread information can be used to simulate truly parallel execution.

4.1.1 Threads and Switch Points in Trace Files

Every operation in a trace file is associated with a thread that has performed it. There is a reason for how thread labels appear in a trace file and why they are arranged the way they are.

When the Instrumented JVM is running an application, it starts multiple threads. The JVM's scheduler rapidly switches between threads performing preemptive multitasking. In this way every thread gets a time frame or a session when it modifies the heap or accesses existing objects. In a trace file these per-thread-sessions are clearly visible, as in a trace file there are batches of operations performed by one thread. The size of these batches varies significantly, usually from few thousands to hundreds of thousands [22]. Another important observation we can make is that there is no obvious pattern in the order threads appear in a trace file. Obviously the reason for both inconsistency in batch length and unpredictable order is the same: the scheduler does not simply follow a turnaround algorithm like round robin, giving tasks that are independent from each other equal time slices, instead we see a result of complex synchronization between threads that happened during the run on the instrumented JVM and was recorded to the trace file. This synchronization is the reason why it is impossible to simply chunk a trace file into smaller pieces and let them run in parallel without errors. If during the original run a thread accessing an object stopped waiting for another thread, GarCoSim should take care of this dependency if the same race condition occurs during the simulation.

Although the inconsistent length of the operation batches and the irregular order of the treads in the trace files are the results of thread synchronization, they do not necessarily allow finding the exact data that was attempted to be

accessed and caused the switch. But the moment when the switching happens can help us find the data that was the reason for the synchronization. IT is known that both threads on both sides of the switch point were accessing the same object. It is also known that it is important to preserve the order in which these two threads had accessed this object during simulation. However, there are multiple objects satisfying this requirement, and such pairs can be found even around switch points, that were simple time slices in the original run. In the next section this problem is discussed further. It is also discussed if it is possible to solve it at all. Later an alternative way to approach the problem is presented.

4.1.2 Recovery of Synchronization vs. Guarantee of Correctness

So far, the structure of trace files was presented, it was also discussed how it represents inter-thread synchronization together with simple time slicing. The problem is, however, to make operations corresponding to a particular threads executable in a truly parallel fashion by physical threads. That means it is required to somehow recover the original synchronization between the treads by looking only into a trace file or into an object tree that is changing as the simulator progresses through a trace file. It is important to keep in mind that the input data is very limited: the workflow of the original algorithm and the logic behind it are unknown, original data that was ma-

nipulated is not accessible, all that is known from trace files is how much memory was allocated for an object, how this object was referenced to/from others and how these objects were accessed during the execution. It is not even known when exactly these operations happen, only their order is available. In a situation of such limited information, it is not feasible to recover the original logic of the application. The knowledge of *what* happened does not give us an understating of *why* it happened. If the original logic is not recoverable, then original synchronization is not recoverable either, because it is unknown which operation of a given thread caused it to stop execution, wait for the time it was inactive, and what brought it back to active state. Neither it is known that such a synchronization happened at all and the switching point was not a simple time slice.

However, it is obvious that the operations in a trace file cannot be shuffled arbitrarily. As it is a sequence of storing and accessing, reads and writes, therefore the operations in it still have to make sense: information cannot be read before it is written, an object cannot be accessed before it is allocated, etc. Thus instead of speculating which of these input-output pairs could have caused a synchronization break (if it had taken place at all), it is possible to find *all* of them and guarantee that the data flow in each particular case was not violated. For the object tree this practically means that object allocation, access and modification happen in the same order and thus they are becoming garbage at the same point: when the entire sequence of its

modifications and accesses is finished. This property is crucial to this thesis: while substituting a not feasible task of recovering the original algorithm's logic with a feasible task of tracking and preserving correctness of the data, it is possible to make GarCoSim run the same trace files (after some additional post processing; details will be described in the upcoming section) in a truly parallel manner and expect to simulate garbage collections correctly at any point of time.

The last thing that is needed to be discussed is how the tasks of synchronization recovery and data correctness preservation are related to each other. The key difference is that the latter covers *all possible* data dependencies, not only ones that actually had taken place. From this perspective the task of correctness preservation is stricter. That means that a set of dependencies found by this approach is bigger than the set of actual dependencies that had taken place and caused changes in the application run.

4.2 Solution Approach

This section describes the approach that was taken to the problem. First, the method of trace files splitting and parallelization is discussed. Later, a way of execution of separated trace files is presented.

4.2.1 Trace File Parallelization

In the original version of GarCoSim, one single trace file is executed line by line. The sequence of commands is always the same, exactly as they happened during the original run when the trace file was captured. However, it is desired that each thread has its own sequence of commands, so logically that means that each thread needs to have a separate individual trace file. Within each thread trace file, the order of commands remains the same, which means that operations performed by same thread do not need to be synchronized, even if they come in different blocks in original trace file, as long as these blocks belong to the same thread. The case where a synchronization is needed is a pair of operations where each operation belongs to a different thread. In this section it is discussed how it is possible to find these pairs and define a data dependency.

As discussed earlier in Subsection 4.1.2, the task is to guarantee correctness of the data. This means a guarantee that values is are read correctly correspond to the values that were written before, in a sequence that cannot be broken: if a value A is written to some variable X, then this value is read several times, and then a value B is written to X, then: 1) all reads of A from X should happen after A was written to X; 2) all these reads should happen before B is written to X; 3) the order of reading of A from X does not matter. Similarly, if there are a number of writes to a variable, they all should happen after the last read of an old value, but before the corresponding reads of new values.

What is not obvious is that unlike reads, writes cannot be shuffled, even if there are no reads in between. The reason is that in GarCoSim writing to an object means a change of reference, objects are referencing each other, such referencing creates an object tree. In parallel to this set of writes another thread can be allocating memory for new objects. As a result, at any moment of the simulation, GarCoSim can report that there is no free memory for new allocations and can trigger a GC. That means that GC can start between any of these writings operations, but if they are shuffled, the topology of the tree is not guaranteed to stay the same: leaves that were not reachable from the rootset anymore (i.e., they are garbage) can now stay connected to the tree, and vice versa, live data can appear cut from the tree. As a result, live data and garbage can be confused, which would defeat the purpose of the simulation itself. Therefore, the object tree should be modified exactly in the order it was modified in the original trace file, meaning all writes should come in the same order. And if so, all reads that come after a series of writings will wait only for the last write to happen. In the end, all the observations above can be summarized in a set of three possible cases:

- All reads (Rs) are waiting for the last write (W) to happen, but the order of Rs themselves does not matter;
- A W is waiting for all previous Rs of an old value to happen;
- A W is waiting for a previous W to happen, even if there were no Rs in between.

Visually this can be depicted as on the following diagram:

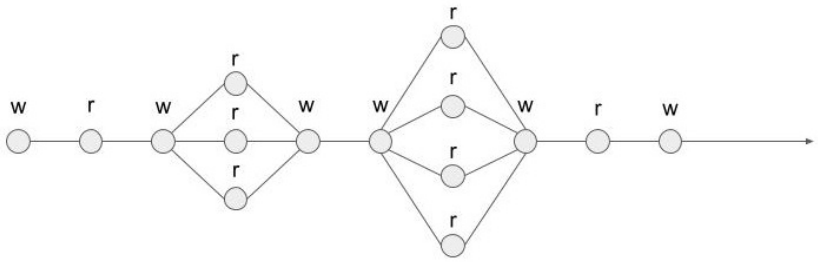

Figure 4.1: Operations on an object over time

Every object, after it is allocated, is being written first. After an initial W there can be either Rs (one or many) or another W, and so on.

Every line on this diagram is a dependency that guarantees data correctness. As it was discussed earlier, every dependency requires synchronization only if it happened between different threads, i.e., the line on the diagram connects two operations performed by different threads. As it is shown on the diagram, the left point of the dependency must be executed before the right one. In other words, the thread that corresponds to the right point of the dependency should be able to monitor the moment when the left one has finished. For that a special command can be added to the trace file of the waiting thread to postpone execution of that particular operation until the left command is executed in the preceding thread. To mark a point in the preceding thread trace file that is required to be executed first, the line number of this command in the preceding thread can be used. There is no

need to add a specific command to the preceding thread trace file. Instead what can be monitored is how far this thread progressed during execution by reporting what line is currently being executed. This information will be available through a shared array and all threads will have access to it. It is important to note, that the shared array itself does not need to be synchronized, as every thread will update only its own current line numbers and access others only for reading. The last question that needs to be discussed is how it is possible to find every dependency in the whole original big trace file. Obviously, it is required to go through every R and W operation and either every time search for a possible dependencies in previous lines or somehow intelligently remember everything that has happened before in order to recover dependencies from that information. Given the size of trace files, searching is not feasible, as it will take too much time. Instead an array can be used, it will record all Rs and Ws to all objects. Values of this array will store thread trace file line numbers of corresponding commands, so once a dependency is found, a corresponding waiting command can be added to the trace file of the waiting thread.

4.2.2 Parallel Trace File Execution

Once all waiting commands are added to corresponding thread trace files for each dependency, they can be executed with a guarantee of data correctness. The simulator will monitor execution of every thread and update a corresponding value in a shared array to let other threads know at what point of

the execution each thread is. When a thread hits a waiting command in its trace file, it is known what line of what thread should be executed before, and will wait until that thread reports passing this point by writing the value of the currently executed line, which is equal or greater than expected. If there were no logical errors during, creation of thread trace files, there should not be any circular dependencies.

Chapter 5

Implementation

This chapter describes implementation details for the thesis. As the code development for the thesis is done in two independent parts, namely trace file splitting in Python and changing GarCoSim code in C, this chapter has two corresponding sections for each of the parts.

5.1 Trace Files Splitting

As discussed in Chapter 4, a solution is needed that will process a trace file line-by-line, splitting it into individual thread trace files and inserting wait commands where necessary. It is desired that this processing program would integrate into the existing process of trace file maintenance. Currently, trace files for GarCoSim are obtained as follows: the instrumented JVM collects a "raw" trace file during actual run of an application, then the "raw" trace file

needs to be handled by by a post-processor. The post-processor is a Python script that reorganizes the input "raw" trace file according to the specification and makes the trace file more readable and easier to maintain. It was decided to implement the trace file splitting procedure as another segment in this pipe: a separate Python script that can take a post-processed trace-file and split it. Because both scripts are implemented in Python, in the future the post-processor and the file splitting script could be combined into a single script.

For the script, it was decided to use Python v2.7. As the script is going to perform numerous array manipulations, the script will be using the NumPy package. NumPy is the fundamental package for scientific computing with Python and is a standard library for many Python distributions [2]. The NumPy package is specifically optimized for numerous array manipulations and is very efficient time-wise and space-wise.

As discussed in Subsection 4.2.1, the parsing script needs to go through the entire trace file and remember all the information required for finding all data dependencies. It needs to remember R and W operations performed by every thread on every object. For that, a table will be used, where objects, threads and operations (Rs and Ws) will be its dimensions; in other words, the size of the table is $N_O * N_T * 2$, where N_O is the number of objects used in the trace file, N_T is the number of operation threads and 2 encodes two op-

erations: reading and writing. The table starts empty (NaN values, because 0 would be a valid reference) and is filled with the line numbers of the trace file that introduce the corresponding operations. The fact that there is no need to add a signal command for every dependency (see Subsection 4.2.1) allows lines of the input trace file to be copied to thread trace files as they are processed, without storing them in memory. This is important, as the trace files are big (from a few to tens of gigabytes). In this way, during the script run only the dependency tracking table needs to be stored in memory.

As a new R or W command for a particular object is read from the trace file that is being split, the script uses the dependency tracking table to find all the dependencies with other operations performed earlier on this object by other threads. In this manner the script traverses the input trace file splitting it into thread trace files and inserting waiting instructions where necessary.

5.1.1 Interpretation of Trace File Commands as Rs and Ws

In Section 4.2 it was discussed that objects undergo two types of operations: Rs and Ws. Trace files are more complex and contain multiple operations that can access (which corresponds to Rs) or modify (which corresponds to

Ws) objects [24].

As many of these instructions access or modify objects, it makes sense to group them by categories — Rs and Ws — depending on what each instruction is intended to do. As the result, the Rs category (accessing) includes: "r" (for O_j in "Read a primitive or a reference field from an object"), "c" (for O_l in "Store/write a static object reference field into a class object") and "w" (for O_l in "Store/write an object reference field into an object"). The Ws category (modifying) includes: "w" (for P_j in "Store/write an object reference field into an object"), "s" (for O_j in "Store/write a primitive field into an object") as well as "a", "+" and "-" instructions, which are included into the category as they change the way the simulator processes them. It is important to note that "w" instructions are interpreted as modifying (W category) for the parent object (P_j) and accessing (R category) for the referenced object (O_l) at the same time. In other words, a single w operation will create separate separate records in the dependency tracking table for both parent (P_j) and referred (O_l) objects and can potentially cause two different synchronizations for each of them.

Given these rules of trace file instructions interpretation, it is possible to keep history of access and modification of all objects through the entire trace file using all the variety of instructions in it.

5.2 Parallel Trace Files in GarCoSim

The current implementation of GarCoSim is single threaded. The *main* function is the only running thread; it (directly or indirectly) creates all the necessary objects (simulator object, memory manager, collector, etc.), it reads the trace file line-by-line, reconstructing an object tree in a simulated heap, and it switches to garbage collection when necessary.

Multi-thread execution has a few significant distinctions. First of all, it is the threads themselves. Threads have to be created by the operating system and managed independently by a scheduler. To implement threads, POSIX threads or the *pthreads* library is used. The simulator now accepts multiple trace files, creates a separate thread for each of them and allows them concurrently, in parallel, to execute instructions modifying the common heap.

Second, the simulator requires an internal data structure to synchronize R and W operations according to waiting signals that were added during the trace file splitting. As discussed earlier, trace file line numbers are used to mark points in the trace files where execution of the current thread needs to be postponed until the specified line of the specified other thread trace file is executed. For this purpose, every thread will report the line of its thread trace file that it is processing and as soon as the awaited value (or greater) is reported, the waiting thread can safely continue execution until the next

waiting point.

The third difference is the allocation of a new object. As now multiple threads are executing their own instructions, it might happen, that two or more threads will simultaneously request allocation of on object on the simulated heap. Such a collision can lead to unpredictable results and incorrect allocation. For that reason it is required to make the allocation an atomic operation, meaning that only one thread is allowed to perform it at a time.

The fourth difference is the garbage collection. Garbage collection requires that while it is happening, no changes can be made to the collected area of the heap. If the entire heap is being collected, that means that a *stop-the-world* situation will accrue, when every manipulation of the data in the heap is paused until the GC is finished. In single-threaded execution of GarCoSim it happens naturally: a single thread can either read and execute trace file, modify the heap, or perform a GC. In multi-threaded execution it should be explicitly controlled: all threads need to stop execution before the GC can start.

According to the four main points described above, the changes were made to the GarCoSim:

- The executable file accepts multiple thread files as arguments; a separate pthread is created to maintain each of them;

- Each thread processes, corresponding thread trace file, passing the processed command to the simulator object, until a waiting command is reached in the trace file. Every thread reports the number of the currently executed line to a shared array; when a waiting operation is reached, the thread pauses execution until the awaited value (trace file line) is not reported by the awaited thread;

- The allocation operation is considered atomic and is protected my a mutex. Two (or more) threads cannot call allocation of an object to the simulated heap simultaneously;

- Garbage collection requires all threads to pause execution for the time of the GC.

The following chapters will describe the observations, discuss results of the implementation and describe the future work.

Chapter 6

Results Analysis: Verification and Validation

This chapter describes results of the thesis and the implemented solution. The first section discusses what trace files were used for the result analysis. In the second section The performance of the trace file splitting script is discussed. Third section describes how the modified GarCoSim has performed while running thread trace files.

It is important to note, that the problem presented in this work is very specific and to some degree unique: in the original implementation of GarCoSim, the trace files are collected by the instrumented JVM (or generated by the trace file generator) and contain information about individual threads and their activity during the original run. This information, however, is not

used by the simulator, instead all actions are executed in a single-threaded manner. This thesis addresses this problem by presenting a solution of how this information can be used to make the simulation in GarCoSim truly parallel. As the problem and its solution are so specific, it is hard to find any alternative methods that attempt to solve the same or a similar problem and to compare the presented solution with them.

6.1 Trace Files Used for Validation

As discussed in Section 2.4, the trace file generator, is a part of the GarCoSim framework, is a convenient tool for creating trace files for this thesis. These trace files will have desired properties (length of the file, number of operation threads), plus they will be correct and realistic (maintain objects as a real application would).

For this thesis a number of trace files were generated. The length of the trace files is an important parameter that determines for how long the simulated application runs and how many objects (and, as a result, memory) it is using. For this thesis a set of trace file of following lengths were generated: 500,000 (500K), 1,000,000 (1M), 3,000,000 (3M), 10,000,000 (10M) and 20,000,000 (20M). All these trace files have 10 threads operating.

The number of operating threads is another important characteristic. It

shows how many threads are started for the simulated application. The generated trace files are split into a corresponding number of thread-trace files by the splitting script. For the purpose of this thesis a set of trace files with following numbers of threads were created: 5, 10 and 30. The length of all trace files in this case is fixed to 10,000,000 (10M).

Using these two sets of trace files, we investigate how the parallel execution can be different for applications of a different scale (with more or fewer threads active and with bigger or smaller heap required for the run).

Another important characteristic of the experimental setup is how much memory the simulator is given to operate. In other words, what heap size is simulated. The heap size should be enough for the simulation to be finished: GCs should provide enough space for new objects to be allocated. Too large a heap, on the other hand, will provide enough space for all allocations without the necessity of collecting garbage. Intermediate GCs are important to analyze correctness and performance of a parallelized simulation. In other words, the memory pressure should be enough to make GCs happen, but the heap size should be enough to let the simulation finish successfully. The size of the heap was picked experimentally for every trace file with an intention to obtain 5-7 GCs during the simulation. Table 6.1 shows the heap size values that were were selected according to how many operations a trace file has.

Number of operations	Heap size
500000 (500K)	675KB
1000000 (1M)	1375KB
3000000 (3M)	4300KB
10000000 (10M)	14750KB
30000000 (20M)	20MB

Table 6.1: Size of heap for different trace files

Such a set of trace files allows the investigation of parallelized simulation in different circumstances. The same set of trace files is used to test splitting performance.

6.2 Trace Files Splitting Performance

As it was described in Subsection 4.2.1, the trace file splitting script is designed to process the original trace file once. That means that the time complexity for the trace file splitting is $O(n)$ and the execution time grows linearly depending on the size of the input file. Experiments confirm that the execution time for the splitting script grows linearly as the number of operations in the input file increases. Figure 6.1 demonstrates these results.

As it is stated in Section 5.1, the splitting script is using a table to track all the dependencies. The size of the table is $N_O * N_T * 2$, where N_O is the number of objects used in the trace file, N_T is the number of operation threads and 2

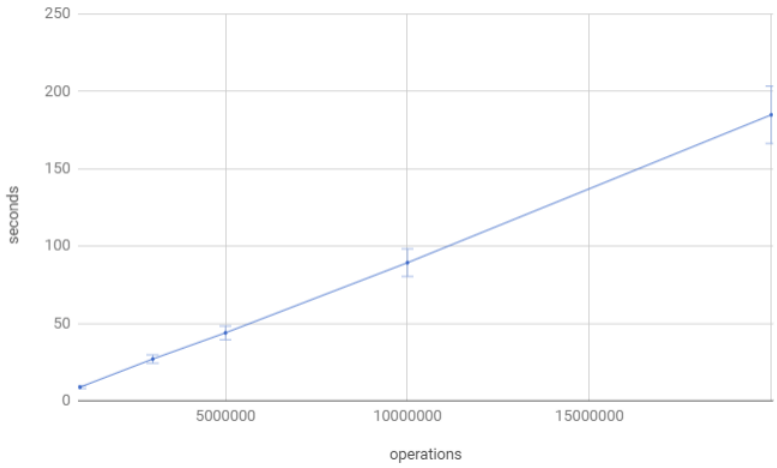

Figure 6.1: Trace file splitting time depending on the number of operations in it

encodes two operations: reading and writing. Space complexity for the trace file splitting is $O(nt)$, where t is number of threads in the input trace file. Figures 6.2 and 6.3 demonstrate linear growth of memory footprint for trace files of various lengths with a fixed number of operating threads and for trace files of a fixed length but with different numbers of operating threads.

6.3 Thread Trace Files Execution

This section describes how the modified GarCoSim runs the individual thread files after splitting. Firstly, the correctness of the data is discussed: it is important that parallel execution of the split thread files produces the same results as a sequential execution of the original trace file. Later in this section,

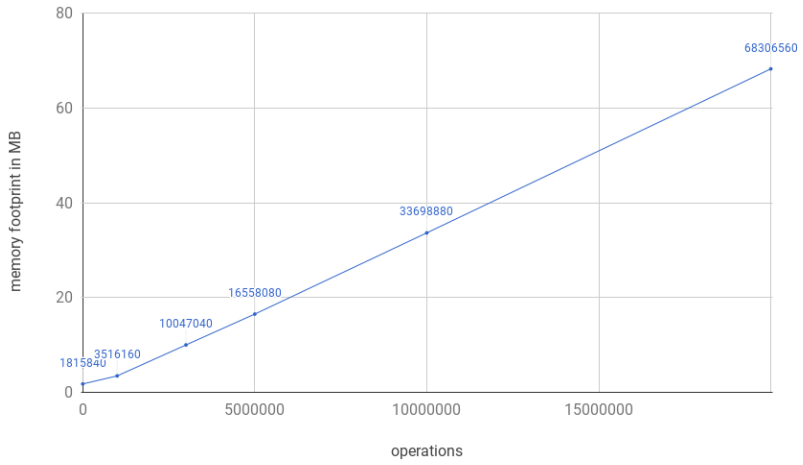

Figure 6.2: Memory used by trace file splitting depending on the number of operations in it

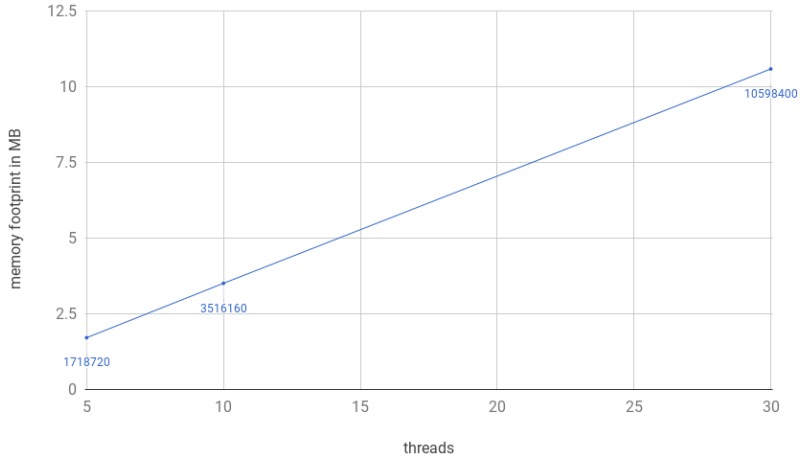

Figure 6.3: Memory used by trace file splitting depending on the number of operating threads

the performance of parallelized GarCoSim is discussed. In this subsection total execution time of the sequential and parallel simulations is compared. Lastly, Garbage Collections in parallel and sequential executions are compared. Different metrics are used to show how splitting and running thread trace files in parallel influenced GCs.

6.3.1 Validity of the Data

The main purpose of the simulation is to reconstruct the object tree as it was created during the original run on the instrumented JVM. Unlike the original run though, we would like to allow GC to happen at an arbitrary moment and still provide correct results. In this context, correctness means that at any moment live data and garbage can be identified correctly. For a single-threaded execution this means a reconstruction of an object tree exactly as it was created during the original run in the instrumented JVM. For multi-threaded execution, on the other hand, this is no longer true, as the objects are now allowed to be created in arbitrary order; their creation is independent, meaning the structure of the object tree cannot be known in advance. The only exception to this rule is at the very end of the simulation, when all the operations from the trace file were executed. In this case, a forced final GC should collect all the garbage objects and leave all the live ones, providing as a result the exactly same object tree no matter if it was constructed by a single or multiple threads. To summarize, because of the non-deterministic nature of parallel execution, the only point where it is

possible to compare it to a sequential execution is the very end of simulation, that is followed by a forced GC. After the forced final GC, it is important to be sure that the parallel execution of the split thread files produces:

- The same results each time. Despite the fact that multiple threads are running in parallel in a non-deterministic manner, synchronization points prevent race conditions and the final result is always the same.
- The same results as a sequential execution of the original trace file. Although parallelization of the simulation makes the entire process non-deterministic, the final object tree of live objects should remain the same. In other words, each time parallel execution happens, the threads can potentially perform allocation, accessing and abandoning of the objects in different order, but all possible execution paths should lead to the same result: live data should remain live, garbage should be detectable as garbage.

These criteria were used to check if the implemented solution produces correct results. Test runs showed that the forced GC at the end of the multi-threaded simulation leaves the same number of live objects at the end of the single-threaded simulation. This means that no objects were added to live data by mistake. Another observation is that, during the run there were no attempts to access objects, that were collected. In addition, no objects were mistakenly collected as garbage. These two observations allow the conclusion, that multi-threaded simulation produces correct results.

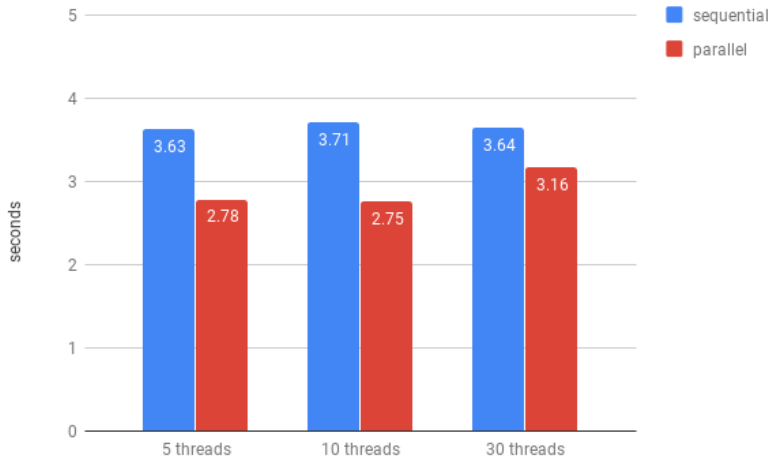

Figure 6.4: Sequential vs. parallel execution time in seconds for trace files with 5, 10 and 30 operating threads

6.4 Performance Comparison

As there is a change from sequential execution to parallel, it is expected that overall execution time will change. Test runs showed that the multi-threaded execution is consistently faster than the single threaded one. The results are similar for trace files with various numbers of threads operating (Figure 6.4).

Very similar results were obtained for trace files of various lengths. Figure 6.5 shows the execution time in seconds (log scale) for trace files of different length. Trend lines (linear) are added for convenience. In all cases parallel execution is faster by approximately 20-25%. This is an expected result.

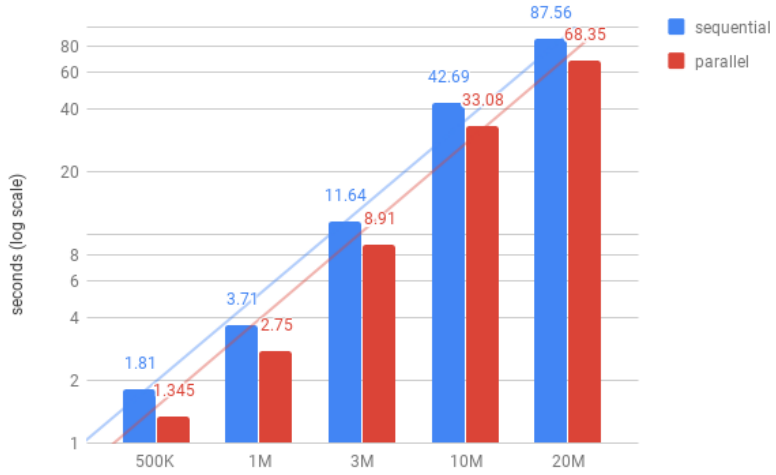

Figure 6.5: Sequential vs. parallel execution time in seconds for trace files of different length

As we discussed in Subsection 4.2.1, operations on the object must come in strict sequence to guarantee data correctness at any moment. Only parallel R operations can be executed. The trace files used for this test were generated with a percentage of read operations equals to 80% (the default value [3]).

6.5 Garbage Collections and Pause Time

Switching from sequential execution to parallel also influences performance of GC. In test runs, the number of GCs and their performance are measured. The results are presented in Figures 6.6 and 6.7. The performance of GCs is measured in objects collected.

	Number of threads		
	5 threads	10 threads	30 threads
sequential			
GCs	6	6	6
min obj collected	5527	5841	5928
ave obj collected	10204.33	10501.33	10948.17
max obj collected	12553	12772	13566
parallel			
GCs	6	6	6
min obj collected	5528	5843	5930
ave obj collected	10204.33	10466.33	10434.33
max obj collected	12552	12551	13181

Figure 6.6: GC statistics for trace files with 5, 10 and 30 operating threads

	Length of trace file				
	500K	1M	3M	10M	20M
sequential					
GCs	7	6	6	6	5
min obj collected	3055	5841	16306	53635	110572
ave obj collected	5857	10501.33	29328.67	95339.33	187691.6
max obj collected	7100	12772	35702	115264	231447
parallel					
GCs	7	6	6	6	5
min obj collected	3058	5843	16302	53637	110569
ave obj collected	5860.1	10466.33	28216.33	93901.67	178368.2
max obj collected	7104	12551	35685	112165	231231

Figure 6.7: GC statistics for trace files of different length

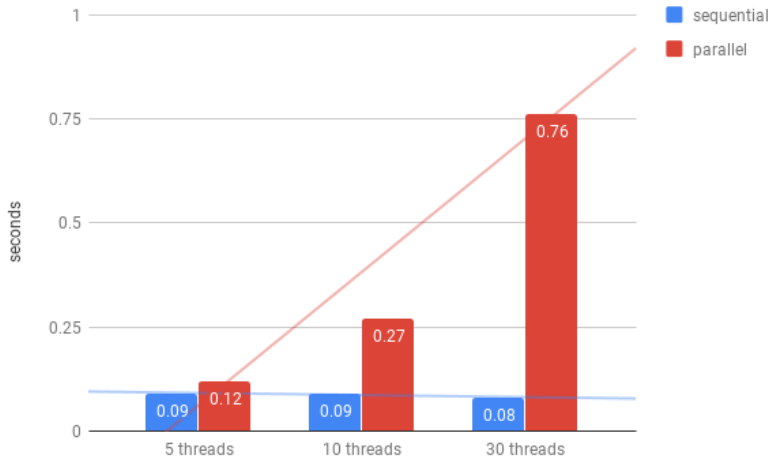

Figure 6.8: Average GC time in seconds for trace files with 5, 10 and 30 operating threads

Interestingly, not only are the number of GCs the same, but also GC statistics are very similar. This indirectly indicates, that entire parallel run is similar to a compressed version of the sequential run, where similar events happen faster. As in was shown in the previous Section, in the case of a parallel execution, the simulation accelerates by approximately 20-25%. Apparently this acceleration is valid for all events during the simulation.

Another important characteristic of GC is pause time. It is interesting to compare average GC time in sequential and parallel executions. The results for trace files with different number of threads operating are shown in Figure 6.8.

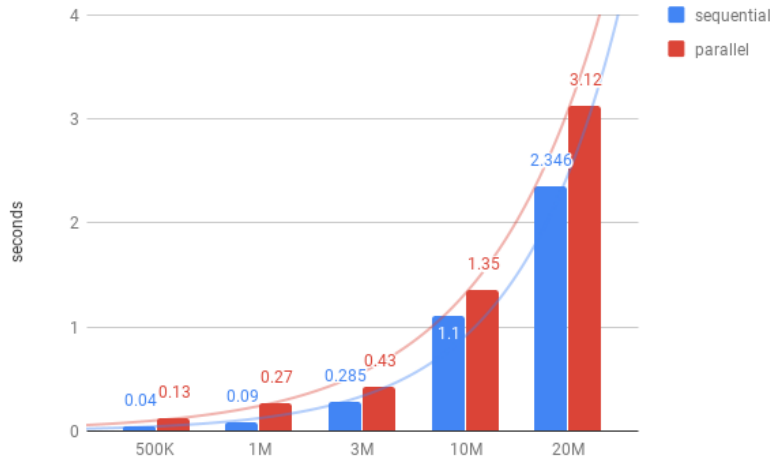

Figure 6.9: Average GC time in seconds for trace files of different length

In the case of a sequential execution it does not matter how many operating threads are there in the trace file. The execution is performed by one thread, once the thread hits an memory allocation error, it starts a GC. All the instructions in the trace file that not executed yet, will wait until the only thread finishes the GC and returns to trace file instructions execution. For the parallel execution, there are multiple threads actually operating in the moment when one of them fails to allocate memory for a new object as there is no more free space left. The thread that hits a memory allocation error must signal others to pause execution for the GC. Other threads need to finish the operations they are working on in order to allow a GC to start. The more threads there are, the longer time this synchronization requires.

The same comparison for the trace files of various length but with the same

number of threads (10 by default [3]). Figure 6.9 shows that in this case the slowdown is consistent.

Chapter 7

Conclusions and Future Work

The main purpose of this work was to add a multi-thread support to GarCoSim and to enable it run a modified trace file by a multiple threads. The work was divided into two parts:

- A parsing script that can process a trace file into individual thread trace files, that can be executed by individual threads. An important role in the parsing was given to synchronization commands, which are added to trace files to guarantee correctness of the data.
- A modification of the GarCoSim code, that would allow multiple thread files to run in parallel, correctly using synchronization commands added during the parsing.

Test runs showed that the modified version of GarCoSim indeed runs separate files in parallel, producing correct results. Overall execution time is

shorter by approximately 20-25% with a tendency of GCs to happen proportionately faster. Pause time of GCs increases with a growth of operating threads numbers, because these threads are physically running during the parallel execution and need to be stopped for a GC.

The following can be defined for scope of future work:

- Although the execution of the simulation in GarCoSim is truly parallel and the results are correct, the scale of actual parallelization can be investigated further. In this thesis the task of original synchronization recovery is substituted with the task of the guarantee of correctness (see Subsection 4.1.2). Apparently, such a substitution creates conditions for the execution that are too strict and do not allow a more accurate parallelized simulation. The reason for this is in how the trace files are captured. An actual application running on the JVM can be executed in various different ways: different parts of the code can be executed in different orders by different threads. Synchronization mechanisms like mutexes and semaphores guarantee that the synchronized code will not be executed simultaneously by different threads, but do not deal with the order in which threads happen to execute this code. However, in the solution presented in this thesis, this particular order seems to be preserved by the set of restrictions that guarantee data correctness. The trace file represents one particular way the application ran and that was captured by the instrumented JVM. Being split according

to threads in it and parallelized, the trace file still holds and repeats this one possible execution scenario. Under these conditions, parallel execution can be faster (as reading operations can happen in parallel), but not fundamentally different from the sequential execution.

- The approach presented in this work provides a technically correct execution. However, the parallelism achieved in simulation is limited because the trace file holds only the information about one particular run during which it was captured. It is possible that processing multiple instances of the same application run (multiple trace files collected by the instrumented JVM running multiple runs of the same application) and combining these results can produce a less restricted, therefore a better parallelized solution.

Bibliography

[1] *IBM knowledge center*, http://www.ibm.com/support/knowledgecenter, Access date: June 1, 2018.

[2] *NumPy: the fundamental package for scientific computing with python*, http://www.numpy.org, Access date: June 14, 2018.

[3] *Tracefilegen: automatic generation of basic memory management operations*, https://github.com/GarCoSim/TraceFileGen, Access date: June 1, 2018.

[4] Eva Andreasson, *Reinforcement learning for a dynamic java virtual machine*, 2002.

[5] Eva Andreasson, Frank Hoffmann, and Olof Lindholm, *To collect or not to collect? machine learning for memory management.*, Java Virtual Machine Research and Technology Symposium, 2002, pp. 27–39.

[6] Chris Bailey, Charlie Gracie, and Karl Taylor, *Garbage collection in WebSphere Application Server V8, Part 1: Generational as the new default policy*, IBM WebSphere Developer Technical Journal (2011).

[7] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham, *Controlling garbage collection and heap growth to reduce the execution time of java applications*, ACM Trans. Program. Lang. Syst. **28** (2006), no. 5, 908–941.

[8] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis, *Garbage-first garbage collection*, Proceedings of the 4th international symposium on Memory management, ACM, 2004, pp. 37–48.

[9] Sylvia Dieckmann and Urs Hölzle, *A study of the allocation behavior of the specjvm98 java benchmarks*, European Conference on Object-Oriented Programming, Springer, 1999, pp. 92–115.

[10] Damien Doligez and Xavier Leroy, *A concurrent, generational garbage collector for a multithreaded implementation of ml*, Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1993, pp. 113–123.

[11] John R Ellis, Kai Li, and Andrew Appel, *Real time, concurrent garbage collection system and method*, February 11 1992, US Patent 5,088,036.

[12] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya, *ifogsim: A toolkit for modeling and simulation of resource*

management techniques in the internet of things, edge and fog computing environments, Software: Practice and Experience **47** (2017), no. 9, 1275–1296.

[13] Matthew Hertz and Emery D Berger, *Quantifying the performance of garbage collection vs. explicit memory management*, ACM SIGPLAN Notices, vol. 40, ACM, 2005, pp. 313–326.

[14] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng, *The garbage collection advantage: Improving program locality*, SIGPLAN Not. **39** (2004), no. 10, 69–80.

[15] Richard Jones, Antony Hosking, and Eliot Moss, *The garbage collection handbook: the art of automatic memory management*, Chapman & Hall/CRC, 2011.

[16] Richard Jones and Rafael Lins, *Garbage collection: algorithms for automatic dynamic memory management*, vol. 208, Wiley Chichester, 1996.

[17] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo, *Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd*, ACM Transactions on Embedded Computing Systems (TECS) **16** (2017), no. 5s, 134.

[18] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The java virtual machine specification: Java se 8 edition*, Pearson Education, 2014.

[19] Shun Long and Michael O'Boyle, *Adaptive java optimisation using instance-based learning*, Proceedings of the 18th Annual International Conference on Supercomputing (New York, NY, USA), ICS '04, ACM, 2004, pp. 237–246.

[20] Feng Mao and Xipeng Shen, *Cross-input learning and discriminative prediction in evolvable virtual machines*, Code Generation and Optimization, 2009. CGO 2009. International Symposium on, IEEE, 2009, pp. 92–101.

[21] Feng Mao, Eddy Z Zhang, and Xipeng Shen, *Influence of program inputs on the selection of garbage collectors*, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, 2009, pp. 91–100.

[22] Konstantin Nasartschuk, Marcel Dombrowski, Tristan Basa, Mazder Rahman, Kenneth Kent, and Gerhard Dueck, *Garcosim: A framework for automated memory management research and evaluation*, Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools (ICST, Brussels, Belgium, Belgium), VALUETOOLS'15, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 263–268.

[23] Simon Ostermann, Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer, *Groudsim: an event-based simulation framework for computational grids and clouds*, European Conference on Parallel Processing, Springer, 2010, pp. 305–313.

[24] Md Mazder Rahman, Konstantin Nasartschuk, Kenneth B Kent, and Gerhard W Dueck, *Trace files for automatic memory management systems*, Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, vol. 2, IEEE, 2016, pp. 9–12.

[25] Lorenza Saitta and Jean-Daniel Zucker, *Abstraction in artificial intelligence and complex systems*, vol. 456, Springer, 2013.

[26] Ryan Sciampacone, Peter Burka, and Aleksandar Micic, *Garbage collection in WebSphere Application Server V8, Part 2: Balanced garbage collection as a new option*, IBM WebSphere Developer Technical Journal (2011).

[27] Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos, *Intelligent selection of application-specific garbage collectors*, Proceedings of the 6th international symposium on Memory management, ACM, 2007, pp. 91–102.

[28] Masashi Sugiyama, *Statistical reinforcement learning: modern machine learning approaches*, Chapman and Hall/CRC, 2015.

[29] Richard S Sutton and Andrew G Barto, *Reinforcement learning: An introduction*, vol. 1, MIT press Cambridge, 1998.

[30] Csaba Szepesvári, *Algorithms for reinforcement learning*, Synthesis lectures on artificial intelligence and machine learning **4** (2010), no. 1, 1–103.

[31] Christopher J. C. H. Watkins and Peter Dayan, *Q-learning*, Machine Learning **8** (1992), no. 3, 279–292.

[32] Ming Wu and Xiao-Feng Li, *Task-pushing: a scalable parallel gc marking algorithm without synchronization operations*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, IEEE, 2007, pp. 1–10.

Vita

Candidate's full name: Andrii Kuch

University attended:
Sevastopol State Technical University, 2002-2007, B.Sc. in Computer Science

Publications: None

Conference Presentations:
Poster "Application of machine learning techniques in balanced GC", 2016 Research Expo, Fredericton, NB, Canada