JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Johannes Scheibl, BSc.

Submitted at
Institute for System Software

Supervisor
o. Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Co-Supervisor
Dipl.-Ing. Thomas Schatzl

May 2021

Work Stealing in the HotSpot VM

Master's Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

The Garbage-First (G1) garbage collector relies on a lock-free work stealing algorithm to distribute workload among multiple CPU cores during a garbage collection's evacuation phase. G1 employs memory barriers to guarantee the correctness of the algorithm's implementation. However, such barriers come with a negative impact on performance. Hence, the goal of this thesis is to implement two alternative work stealing algorithms that supposedly need fewer memory barriers than the current implementation in G1 does. After an introduction to the G1 garbage collector, lock-free programming and work stealing, this thesis describes the implementation of two previously selected work stealing algorithms into the G1 garbage collector. Finally, benchmarking these implementations found that they are competitive with the current work stealing implementation. Although, not to a degree that would justify to exchange the currently used algorithm for a new one.

Zusammenfassung

Der Garbage-First (G1) garbage collector beruht auf einem Lock-freien Work Stealing Algorithmus um Arbeit, während der Evakuierungsphase einer Garbage Collection, auf mehrere CPU Kerne verteilen zu können. G1 setzt Memory Barriers ein, um die Korrektheit dieses Algorithmus zu garantieren. Jedoch gehen solche Memory Barriers mit einer negativen Auswirkung auf die Performance einher. Infolgedessen ist das Ziel dieser Arbeit zwei alternative Algorithmen zu implementieren, welche vermeintlich weniger Memory Barriers benötigen als die aktuelle Implementierung in G1. Nach einer Einführung in G1 garbage collection, Lock-freier Programmierung und Work Stealing beschreibt diese Arbeit die Implementierung zweier, zuvor ausgewählter, Work Stealing Algorithmen in den G1 garbage collectior. Schlussendlich zeigten die Vergleiche dieser Implementierungen, dass sie in der Lage sind es mit der Performance der aktuell verwendeten Implementierung aufzunehmen. Jedoch nicht in einem Maße, welches einen Austausch des aktuell verwendeten Algorithmus gegen einen der neuen rechtfertingen würde.

Contents

Chapter 1

Introduction

In this chapter, we describe the motivation, problem domain and our solution approach for this thesis. Additionally, we present some work related to this thesis.

1.1 Motivation

As a vital component of *OpenJDK* [31], the *HotSpot* virtual machine provides a diverse number of garbage collectors that provide automatic dynamic memory management. This thesis focuses on the *Garbage-First (G1)* [8][28] garbage collector (GC).

G1 is a stop-the-world garbage collector, pausing the application while reclaiming memory. Similar to several other OpenJDK garbage collectors, during garbage collection G1 evacuates objects in parallel to decrease latency compared to single-threaded collectors. However, this comes with the challenge of evenly distributing work among all available threads. G1 currently implements the Arora, Blumofe and Plaxton [4] (*ABP*) lock-free work stealing algorithm from 2001 to provide workload distribution. Since ABP's implementation into G1, hardware changed towards an increased number of computing units and a significant amount of research has been done in the area of work stealing. This lead us to the assumption that an alternative work stealing algorithm might be able to outperform the currently used ABP implementation in G1.

This thesis aims to replace ABP with an algorithm that provides shorter garbage collection pause times.

1.2 Problem

The ABP algorithm ensures sequential consistency through the use of *memory barriers* [20]. Memory barriers limit compiler instruction reordering and a modern processor's ability to reorder memory operations, which makes them rather costly. The current trend of potential increased parallelism with an ever-growing amount of available cores in a CPU raises the relative costs of those memory barriers. Furthermore, single core performance had been increasing very slowly for years now.

1.3 Solution Approach

Since publication and implementation of the ABP algorithm there has been done a significant amount of research in the area of work distribution algorithms. This

thesis selects two promising approaches as candidates to implement them in the G1 garbage collector for use in the evacuation phase. Finally, we evaluate these two algorithms against G1's ABP implementation.

Michael et al. [22] combine the concept of *idempotence* of work with traditional work stealing. Idempotence of work assumes that the work may be executed more than once, instead of merely once, trading some repeated work with work stealing efficiency. *Idempotent work stealing* seems particularly suitable for garbage collection, since there it is already common for work stealing tasks to be scheduled more than once. As a consequence, it trades some memory barriers against duplicated steals and additional checks which indicate if a task has already been completed.

Wu and Li [44] on the other hand suggest to distribute workload not by stealing, but by passing it to other threads. The most interesting part about this approach is that it does not need memory barriers for transmitting tasks between threads. This is accomplished by maintaining two unidirectional communication channels between each peer thread. However, it requires a more complex termination algorithm, as threads are not able to tell by themselves whether their work is done, and if they may terminate.

1.4 Related Work

This section presents related work stealing improvement strategies which we also considered during our algorithm selection.

Qian et al. [36] identified that attempted stealing frequency, performance impact of failed steals and whether a task can profit from work stealing are three factors which affect work stealing. They propose *SmartStealing* which decides during execution if stealing is beneficial and disables it otherwise. The algorithm does not steal when no other tasks are running. It randomly selects one victim and terminates if a steal fails. Should a steal succeed, it continues to steal from the same victim as long as it has tasks. This optimization changes the policy when and how the algorithm performs work stealing. This is orthogonal to the problem we want to investigate.

Suo et al. [41] found two problems within HotSpot's garbage collection. The *Parallel Scavenge* collector employs an unfair mutex lock to to protect the global GC task queue while balancing load among GC threads. The second problem is about ineffective work stealing and therefore delayed termination, caused by an inadequate coordination between the JVM and the underlying operating system. The authors propose to deactivate operating system load balancing and to pin GC threads to cores, to cope with the load imbalance caused by the unfair mutex lock. To solve the inefficiency during work stealing, Suo et al. suggest a stealing policy that dynamically balances the amount of steal attempts with regard to the number of active GC threads. Additionally, they show an alternative semi-random work stealing algorithm. Again, the problems solved by Suo et al. operate on a different level than we look at in this thesis.

Dijk and van de Pol [43] suggest to split the work stealing deque in order to overcome two of its major drawbacks: "1) local deque operations require expensive memory fences in modern weak-memory architectures, 2) they can be very difficult to extend to support various optimizations and flexible forms of task distribution strategies needed many applications, [...]" [1]. Dijk and van de Pol introduce concurrent deques which are split into a public and a private section. Whereas the split point separating those sections can be modified in non-blocking manner. HotSpot on the other hand uses a fixed separation of public and private buffers.

1.5 Structure of this Thesis

Chapter 2 gives some background information on OpenJDK and an introduction to the Garbage-First (G1) garbage collector. Next it provides some fundamentals on lock-free programming before introducing work stealing in general and the work stealing implementation currently used in the Garbage-First garbage collector.

Then Chapter 3 presents the implementation of the two selected work stealing/distribution algorithms in the G1 garbage collector evacuation phase.

Next, chapter 4 describes the experimental setup as well as the method of measurement.

Chapter 5 presents and discusses the benchmark results of the new implementations compared to the current one used in G1.

Finally, Chapter 6 concludes this thesis and gives an outlook for future work.

Chapter 2

Background and Problem Analysis

This chapter explains fundamental concepts which are required for understanding this thesis. It starts with an introduction to OpenJDK, the HotSpot virtual machine and the Garbage-First garbage collector. Then it explains the basic concepts of lock-free programming, followed by an introduction to the fundamentals of evacuation in a garbage collection context. Finally, there is a short introduction to work stealing in general and the work stealing implementation currently used in the Garbage-First garbage collector.

2.1 OpenJDK

The Open Java Development Kit (OpenJDK) is the reference implementation of the Java Platform, Standard Edition (Java SE). It houses several components, like the virtual machine (HotSpot), the Java Class Library and the Java compiler (javac) [31].

The OpenJDK version used as a basis for the implementation of alternative work stealing algorithms, in this thesis, is OpenJDK 12u [25]. It furthermore serves as a baseline for the performance comparisons in later chapters.

2.1.1 HotSpot

HotSpot is a *Java Virtual Machine* [30] implementation, which, apart from a class loader, a bytecode interpreter and supporting runtime routines, also includes two runtime compilers and multiple garbage collectors [31].

A major component of HotSpot is the adaptive compiler [28]. It interprets bytecode and identifies performance choke points or *hot spots* within the application. Instead of further interpreting those performance relevant sections of bytecode, the adaptive compiler compiles them to increase their performance.

HotSpot manages the JVM's memory by providing the following garbage collectors: [28]:

Serial Collector
: A single threaded garbage collector which suspends all application threads while performing a collection.

Parallel Collector
: Can utilize multiple threads for garbage collection. It also needs to suspend

all application threads during a collection.

Garbage-First (G1) Garbage Collector
G1 is a mostly concurrent collector for large heaps that aims to consistently meet pause-time requirements. G1 collects garbage in parallel and can perform certain collection phases without suspending application threads. It is currently the default garbage collector in HotSpot.

Z Garbage Collector
An experimental low latency collector that targets smaller than 10ms stop-the-world pauses independent of heap size. ZGC performs object evacuation in parallel concurrent to the application.

2.2 Garbage-First Garbage Collector

The Garbage-First (G1) garbage collector aims to consistently meet pause-time constraints while collecting large heaps. This is accomplished by utilizing multiple processors in incremental stop-the-world pauses collecting both the young and old generation. As a mostly concurrent collector, G1 can perform certain collection phases without suspending application threads. G1 is currently HotSpot's default garbage collector. These are the reasons for selecting this garbage collector for this thesis.

2.2.1 Fundamental Design

The JDK 12 Documentation [28] describes G1 as "[...] a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating garbage collector". To provide these characteristics, G1 partitions the Java heap into equally sized chunks, the regions. These regions are the units of evacuation and may be garbage collected individually. The young and old generation are such sets of regions. In a single stop-the-world garbage collection pause, G1 always collects the young generation and optionally a set of old generation regions.

G1 performs a global mark phase to identify live objects on the heap, and how they are distributed among the heap regions. As this operation happens across the whole heap, it runs in parallel concurrency to the application and does not introduce a stop-the-world pause [28].

Space reclamation during a young collection uses multiple threads, copying live objects to other regions [27]. G1 keeps track of the pause times during collections and, if necessary, adapts the number of heap regions being evacuated to stay within certain timing constraints. It also collects regions first which it knows, from the marking phase, to contain the most garbage (hence the name Garbage-First). G1 generally has its focus on collecting the young generation, as it usually tends to contain the least amount of live objects. Only if the old generation grows beyond a certain size G1 will collect it in addition to the young generation.

2.2.2 G1 Heap Regions

The Garbage-First garbage collector divides the Java heap space into uniformly sized regions, as shown in figure 2.1. Since Garbage-First is a generational garbage collector, each region is associated with either the young or the old generation. When the mutator allocates objects, G1 generally places them in an eden region in

the young generation. If a region is full, memory will be allocated in a new region of the same kind. Non-empty regions can be either one of the following:

Eden
New objects get allocated into an eden region of the young generation.

Survivor
Objects that survive their first garbage collection are promoted to survivor regions which are part of the young generation.

Old
After data survives a certain amount of collections it gets promoted to the old generation, consisting of old regions.

If an object's size is 50 percent of a region's size, or bigger, it is classified as Humongous [28]. Those objects are allocated into their own region, or several contiguous regions if necessary. Humongous regions are logically assumed to be part of the old generation, because it is too costly to copy humongous objects between regions. G1 implements a technique to reclaim humongous objects without a full marking called *eager reclaim* [32] to facilitate their reclamation.

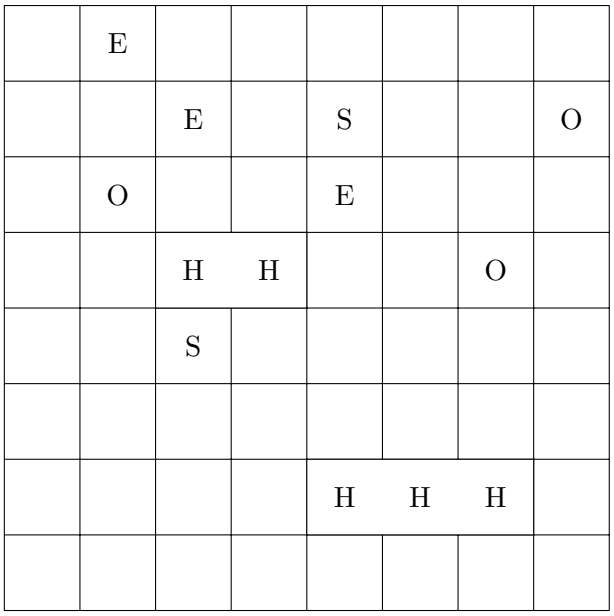


Figure 2.1: G1 heap layout. Showing Eden, Survivor, Old and Humongous regions. Adapted from the G1 tuning guide [28].

2.2.3 Garbage Collection Phases

The Garbage-First collector goes through two major collection phases, called young-only and space-reclamation phase. Figure 2.2 shows a graphical representation of those phases.

The JDK 12 Documentation [28] describes the G1 garbage collection cycle and its two phases as follows:

Young-only phase
G1 keeps track of how many eden regions it is able to collect during one

collection cycle, without exceeding the pause time target [8]. Based on this estimation it initiates a young collection after a certain amount of eden regions were filled through memory allocations. This young collection promotes all live objects from eden to survivor regions. Whereas live objects from survivor regions are either moved to other survivor regions or old regions, depending on the number of collections they have previously survived. As a result, space for new allocations is being freed, while also compacting memory.

When after several young collections the number of live memory in old regions account for a certain percentage of the total heap, the so called *initiating heap occupancy (IHOP)* threshold sets off the preparations for the space-reclamation phase:

Concurrent start
: First the Concurrent Start begins a marking stage in parallel to a young collection's evacuation phase. This stage marks all live objects from the old regions that the Space-reclamation might evacuate later. This marking uses a snapshot-at-the-beginning (SATB) algorithm [45] to provide fast termination.

Remark
: After completing the concurrent mark, the Remark pause takes place. It executes all necessary steps to complete the marking stage. Between Remark and Cleanup, G1 rebuilds the remembered sets by gathering information on which old regions contain most garbage and are therefore best suitable for collection.

Cleanup
: During the Cleanup pause G1 decides, depending on the amount of reclaimable space, whether a space-reclamation phase follows.

Space-reclamation phase
: During the space-reclamation phase the garbage collector will perform mixed collections as long as collecting old regions is beneficial. Those mixed collections are similar to young collections, with the difference that G1 also collects a set of Old regions at the same time.

2.2.4 Evacuation Pauses

Detlefs et al. [8] describe the sequence of an evacuation pause as follows: G1 initiates a stop-the-world pause at appropriate points of execution to make sure that the evacuation of live objects looks like an atomic operation to the mutator threads.

Then the evacuating threads work in parallel to finalize remembered sets, and scan the various roots from the VM for references into the collection set.

The threads then copy every object in the collection set in the Object Copy phase that is transitively reachable from these references into other regions. Copying marks objects as such in the original place to avoid copying a given object multiple times.

2.3 Lock-free Programming

This section presents selected topics on lock-free programming. Those are atomic memory access, compare-and-swap, memory ordering and memory barriers.

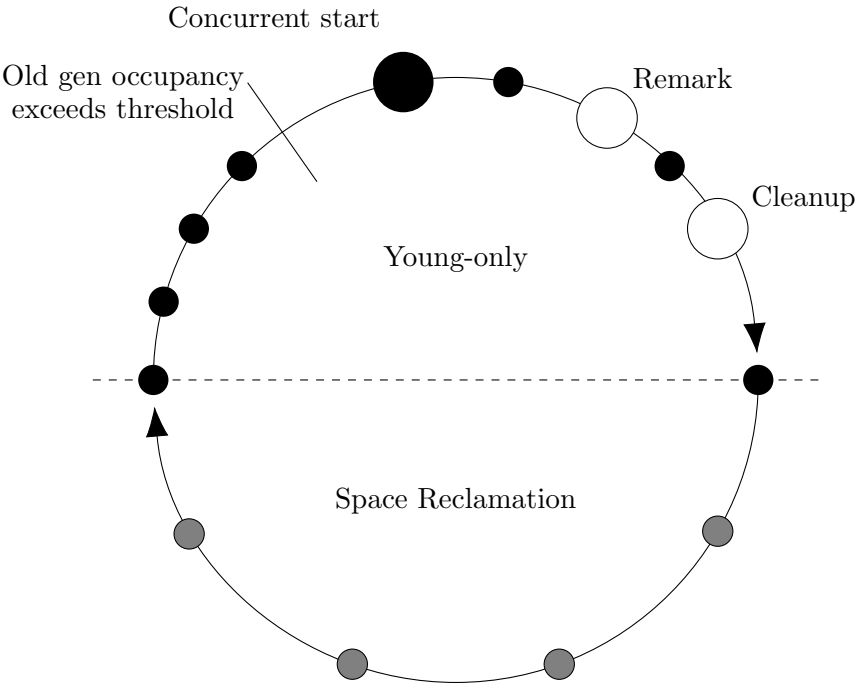


Figure 2.2: G1's garbage collection phases. Adapted from the G1 tuning guide [28].

Herlihy [12] defines that "[...] a concurrent object implementation is lock free if it always guarantees that some process will complete an operation in a finite number of steps [...]". This means that a blocking or very slow thread cannot prevent other threads from making progress, which is important for latency sensitive applications like a stop-the-world garbage collector.

In order to reach such lock-freedom, several different techniques [34] are required. In work stealing multiple readers and writers may access the same memory location. This requires atomic access to guarantee uninterrupted and tear-free reads and writes. The implicit atomicity of access to primitive data types guaranteed by the processor or special read-modify-write instructions provide this. Execution of code in parallel on modern processors typically does not provide sequential consistency. This requires the use of memory barriers to avoid negative effects of memory access reordering.

2.3.1 Atomic Memory Access

> "Any time two threads operate on a shared variable concurrently, and one of those operations performs a write, both threads **must** use atomic operations."

Preshing [35] defines this rule as a precondition for lock-free programming. Such atomic operations are necessary to prevent reading incomplete or *torn* values. Similar effects, called word tearing, load tearing or store tearing [21], can happen when a thread that takes two operations to write one value to memory is interfered by another thread reading the value in between the two write operations. As a result, the reading thread would load part of the new value and part of the old value.

The source code shown in the following chapters and sections assumes that shared primitive data types can be read and written to atomically. The reason

why this assumption holds, although none of the variables are explicitly marked as an atomic, is because CPU architectures usually treat aligned loads and stores of machine word sized elements as atomic. The AMD64 Architecture Programmer's Manual describes access atomicity as follows [2]:

> "Cacheable, naturally-aligned single loads or stores of up to a quadword are atomic on any processor model, as are misaligned loads or stores of less than a quadword that are contained entirely within a naturally-aligned quadword."

The Intel 64 and IA-32 Architectures Software Developer's Manual [15] mentions the following about the alignment of words, doublewords, and quadwords:

> "[...] the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access."

Both citations show that we can treat loads and stores to aligned memory as atomic memory accesses.

Compare-and-Swap (CAS)

Compare-and-swap is a read-modify-write [18] instruction which can atomically perform a read and a write to a memory location. It may be used to synchronize writes from multiple threads to a single memory location, like in a work stealing scenario where several threads manipulate the pointer to a queue's head. Most modern processors provide hardware support for CAS or a similar instruction.

Compare-and-swap takes three arguments: an old value, a new value and a memory location. The CAS instruction compares the old value to the value inside the given location. If they are equal, it swaps the value inside the given location with the new value. Finally, it returns the value initially read from the given location. The CAS instruction succeeds if the return value equals the given old value.

One disadvantage of compare-and-swap is known as the ABA problem which leads a CAS instruction to return successfully although it should not. The issue can happen if another thread modifies the CAS instruction's target location between reading the old value and starting the CAS instruction itself. The following sequence from Dechev et al. [7] shows how this can happen:

1. P_1 reads A_i from L_i
2. P_k interrupts P_1; P_k stores the value B_i into L_i
3. P_j stores the value A_i into L_i
4. P_1 resumes; P_1 executes a false positive CAS

A, B. . . . . Values
P. . . . . Processor
L. . . . Memory location

Employing a version tag [7] [23] is one way to detect and avoid the ABA problem.

2.3.2 Memory Ordering

This section explains reordering of memory accesses performed by the compiler during compilation as well as reordering of memory accesses by the processor.

Compile Time Memory Ordering

When translating high-level source code to machine code, the compiler is allowed to reorder, or even remove, memory operations as long as it does not affect the correctness of a single-threaded program. Compilers typically perform such reorderings when compiler optimizations are enabled to increase program performance. However, reordering of instructions can introduce errors into concurrent programs. To avoid such errors, facilities like explicit compiler barrier intrinsics or special modifiers for variables (e.g. C++11 atomics) can be set to prevent reordering of certain instructions without disabling compiler optimizations in general. However, a compiler barrier is often not enough as it does not prevent memory access reordering by a processor during runtime. But compilers usually do recognize memory barriers, which are used against runtime memory reordering, as a compiler barrier also. Compiler barriers need to be set manually, for example via intrinsic functions.

Runtime Memory Ordering and Visibility

In order to improve performance, many processor architectures utilize out-of-order execution and allow memory accesses, in particular memory loads, to be scheduled in a different order than the machine code defined it. Another example are store buffers [2] [15] [20] which can result in writes to a memory location being visible by other processors in a different order than defined in the machine code. These techniques give processors the ability to hide memory access latency and increase performance.

In parallel programs, memory ordering may introduce unintended effects, since the program order in the machine code differs from the code the processor actually executes. Consider the example (based on McKenney [21] and Mortoray [24]) in listing 2.1. Intuition tells that `thread_1()` will always print `1`. Printing `0` on the other hand is not possible according to program order.

```
1  a = 0
2  is_ready = 0
3
4  thread_0():
5    a = 1
6    is_ready = 1
7
8  thread_1():
9    if (is_ready == 1) {
10     p = a
11     print(p)
12 }
```

Listing 2.1: Conditional Print without Barriers.

When running a compiled version of listing 2.1 multiple times, different sets of executions can result from it. Listings 2.2 and 2.3 show two possible results as pseudo assembler representations. In one version, `thread_0()` executes first, whereas `thread_1()` executes first in the other case. Both examples show that according to the instruction sequence it is not possible for the value `0` to be printed.

```
store a 1
store is_ready 1
load is_ready
load p a
print(p)          // Prints 1
```

Listing 2.2: Thread 0 executed first.

```
load is_ready
// load p a - not executed
// print(p) - not executed
store a 1
store is_ready 1
```

Listing 2.3: Thread 1 executed first.

Assuming `thread_0()` and `thread_1()` execute in parallel where the executing processor of `thread_0()` reorders its instructions and `thread_1()` executes in between `thread_0()`'s writes to its variables. The resulting instruction sequence could look like in listing 2.4. Here we can see that the processor's reordering of the stores in `thread_1()` resulted in a state that should not be possible, according to the code in listings 2.1, 2.2 and 2.3.

The second example in listing 2.5 shows the effect of `thread_1()`'s loads being reordered and interleaved with `thread_0`. Since the processor executing `thread_0()` does not see any dependency between the variables `is_ready` and `a`, it may perform the load of `a` first. This however loads `0` into the variable `a` before `thread_0()` can assign a value to it. This results in `thread_1()` printing the value 0 which is not correct according to the source code in listing 2.1.

```
store is_ready 1
load is_ready
load p a
print(p)          // Prints 0
store a 1
```

Listing 2.4: Stores reordered.

```
load p a
store a 1
store is_ready 1
load is_ready
print(p)          // Prints 0
```

Listing 2.5: Loads reordered.

2.3.3 Memory Barriers

Although memory reordering may be beneficial for performance, the examples in listings 2.4 and 2.5 show how it can affect a parallel program's correctness. To avoid memory accesses to be reordered, processor architectures usually provide so called memory fences or memory barriers [13]. The most restrictive of those barriers is called a full barrier. When placed within the source code, it prevents all instructions, that appear above it, to be executed and completed after the barrier and vice versa. Listing 2.6 shows how the problems from the code in listing 2.1 can be avoided.

```
1  a = 0
2  is_ready = 0
3
4  thread_0():
5    a = 1
6    full_barrier()
7    is_ready = 1
8
9  thread_1():
10   if (is_ready == 1) {
11     full_barrier()
12     p = a
13     print(p)
14 }
```

Listing 2.6: Conditional Print with Barriers.

The barriers added to 2.6 signal the processor that the effect of the write instruction `a = 1` in line 5 must not occur below the barrier in line 6. Additionally, the assignment's execution of `is_ready = 1` in line 7 is not allowed to become visible above the barrier. The same restriction applies to the instructions' effects in lines 10 and 12. Neither the comparison nor the assignment in lines 10 and 12 may be reordered so that their effects are visible past the barrier in line 11.

Listings 2.7 and 2.8 show an approximate translation of listing 2.6 into a possible instruction sequence. The interleaving is the same as in the previous examples from 2.4 and 2.5. However, now the barriers prohibit reordering. The instructions are executed in a way that is consistent to the source code in listing 2.6 and therefore the value `0` will not be printed.

```
1 store a 1
2 full_barrier()
3 load is_ready
4 full_barrier()
5 // load p a - not executed
6 // print(p) - not executed
7 store is_ready 1
```

Listing 2.7: Stores prevented from reordering.

```
1 load is_ready
2 full_barrier()
3 store a 1
4 full_barrier()
5 store is_ready 1
6 // load p a - not executed
7 // print(p) - not executed
```

Listing 2.8: Loads prevented from reordering.

These examples illustrate that memory barriers are sometimes needed for correctness. Nevertheless, barriers are relatively costly since they prevent the processor from gaining performance by reordering memory accesses. As a consequence, they should only be used where really necessary.

Memory Barrier Types

The barriers used in previous examples are full barriers. That means any instruction's effect must not be ordered beyond the barrier. Sometimes however, it makes sense to restrict only certain effects from being reordered. Hence, there exist weaker forms of barriers. They might, for example, prohibit stores from becoming visible across a barrier, but allow loads to take effect before or after them. Therefore, processors usually provide an assortment of memory barriers. The following list shows a few

processor architectures and some of their memory barrier types.

AMD64
LFENCE, SFENCE, MFENCE... [2]

ARM
ISB, DMB, DSB... [13]

SPARC
#StoreStore, #LoadStore, #StoreLoad... [29]

As this list indicates, memory barrier instructions are highly processor architecture dependent. This is unfavorable when developing software that is intended to run on a variety of different processors. Therefore, it is necessary to use abstracted barriers which get converted to native barriers at compile time.

The interface used in HotSpot [26], loosely based on The JSR-133 Cookbook for Compiler Writers [19], defines the following memory barriers:

Fence
A full barrier that prevents any memory access operation's effect above the barrier to be reordered with the effects of the memory operations after the barrier.

LoadLoad
Guarantees that load instructions above the barrier obtain a value from memory before the load instructions below the barrier finish reading from memory.

StoreStore
Ensures that the effects of stores above the barrier are visible in memory before stores below the barrier finish writing to memory.

LoadStore
Guarantees that load instructions above the barrier obtain their value from memory before the store operations below the barrier finish writing to memory.

StoreLoad
Guarantees that the effects of store operations above the barrier are visible in memory before the loads below the barrier finish reading from memory.

In addition to those barriers, there also exist *acquire* and *release* barriers. Those are unidirectional barriers which synchronize around a `load(x)` and `store(x)` combination. The interface [26] offers the example shown in listing 2.9 and describes it as follows:

> "It is guaranteed that if `T2: load(X)` synchronizes with (observes the value written by) `T1: store(X)`, then the memory accesses before the `T1: ]release` happen before the memory accesses after the `T2: acquire[`."

```
1 T1: access_shared_data
2 T1: ]release
3 T1: (...)
4 T1: store(X)
5
6 T2: load(X)
7 T2: (...)
8 T2: acquire[
9 T2: access_shared_data
```

Listing 2.9: acquire / release barrier example [26].

The previous sections showed how memory barriers can prevent memory reorderings that would otherwise lead to parallel programs being incorrectly executed. As a consequence, such memory barriers limit a processor's runtime reordering of memory operations which makes memory barriers much more costly than regular stores or loads. A good work stealing algorithm minimizes the amount of executed memory barriers.

2.4 Evacuating Garbage Collection

Evacuation during garbage collection separates live objects from garbage by copying live objects from one area to other free areas. After this evacuation phase, the source areas contain garbage only, whereas the target areas contain all live objects in a compacted form.

Figure 2.3 shows the heap being divided into two areas [17] called *from space* and *to space*. The heap objects (A, B, C D and F) reside inside those sections. The arrows represent pointers or references to heap objects. Pointers that do not originate from an object are *root pointers*. An object is considered live if a pointer is referencing it. Non-referenced objects are called garbage. Every thread shown in figure 2.3 has its own deque where it stores tasks that spawned while processing a live object.

Every object, that is referenced to by a pointer, creates an evacuation task. Such a task might spawn further tasks, depending on the amount of outgoing references the object has. Threads start by processing root pointers first and traverse the heap objects through the references between them. Figure 2.3 shows an example of an early stage of the evacuation phase. On the left side is the heap section, called *from space*, where live objects are taken from to be copied to the empty *to space*. In this example only the objects A, C and D will be evacuated, since they are being referenced to. Below the heap regions, the threads executing the evacuation and their task queues are shown. The tasks to evacuate C and D are the first ones to be handled, since the root pointers are referencing them, and are therefore already in the queues of threads T_0 and T_2.

Processing D creates the tasks C and A, which thread T_2 appends to its queue, as seen in figure 2.4. Additionally, object D will be marked as evacuated and a forwarding pointer from object D to its copy, D', is created. Similarly, processing the task C creates the object C' and an according forwarding pointer. Since the heap object C initially had two pointers pointing to it, two evacuation tasks spawned while processing this object's task. However, while working on the second C task, thread T_2 is able to recognize that object C has already been copied and will not process

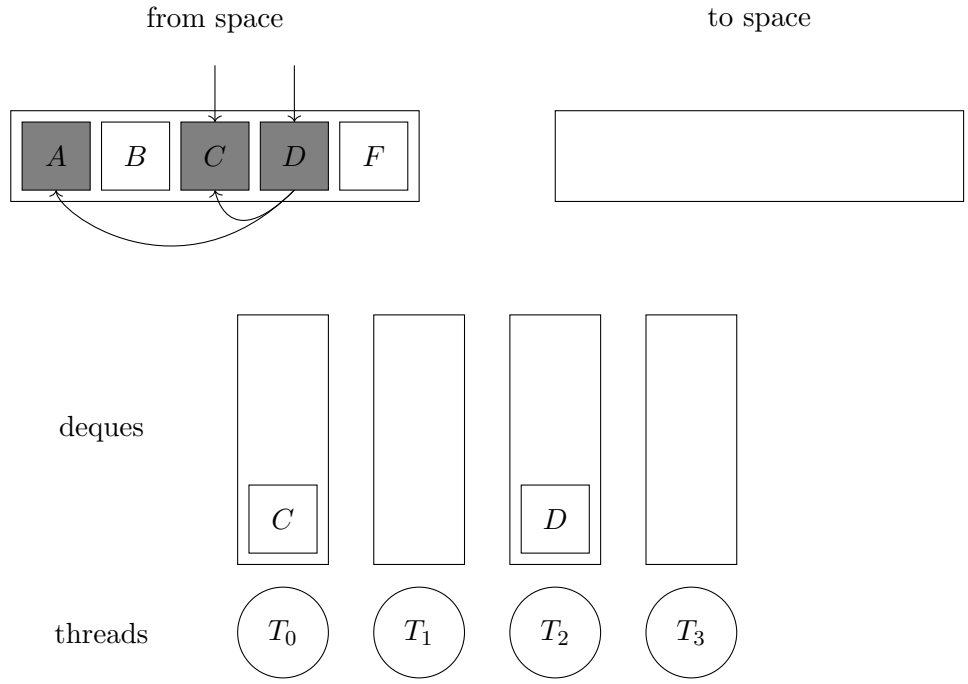


Figure 2.3: Start of the evacuation. All marked nodes will be evacuated.

it any further.

2.5 Work Stealing

Work stealing is a scheduling technique to dynamically balance workload over several workers (e.g. threads). It deals with the issue that some workloads/tasks can neither be accurately predicted nor properly distributed before their execution begins. Reasons might be that tasks spawn an unknown amount of new tasks, or the execution time of each task varies significantly.

Tasks for a given thread are either being processed, or stored in an appropriate data structure, generically called a workpile [37]. Each worker thread has its own workpile to add or take tasks from. Tasks which spawn during the execution of another task will be added to the executing thread's local workpile for later processing.

The workload distribution starts as soon as a thread runs out of tasks, due to its workpile being empty. This thread will then become a thief which takes (steals) tasks from other victim-threads' workpiles. This repeats until the thieves run out of victims they can steal from. Now in order to properly terminate the work distribution stage, it is important to detect that all workpiles are empty and no worker thread can spawn additional tasks.

2.6 Work Stealing in G1

The following sections describe the implementation of the ABP work stealing algorithm for object evacuation in OpenJDK12 [25].

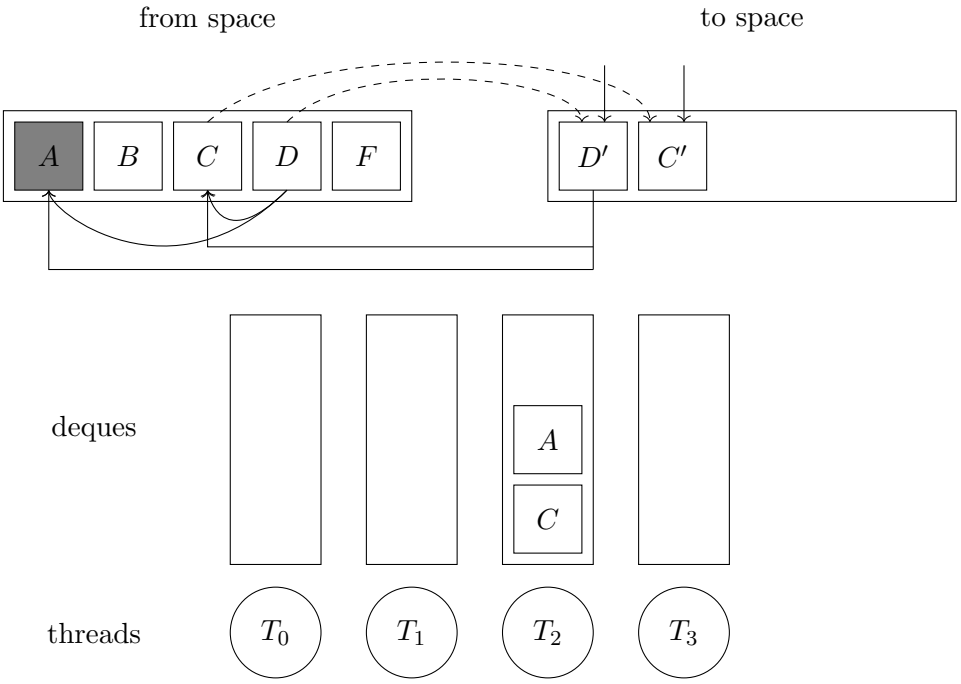


Figure 2.4: Task duplication during evacuation.

2.6.1 Work Stealing Tasks

The tasks in G1's work stealing implementation are pointers which point to heap objects that need to be evacuated. While processing a task, it might spawn new tasks due to heap objects containing pointers to other heap objects. The example in figure 2.5 shows a simple heap where two heap objects are being referenced by root pointers. When object 3 is being evacuated, it spawns two new tasks. Those new tasks will evacuate objects 0 and 2 respectively.

Since object 2 is referenced by two root pointers, there will be two separate tasks to evacuate it. Should the thread executing one of those tasks observe that object 2 already got evacuated (similar to section 2.4), it will not pursue this task any further.

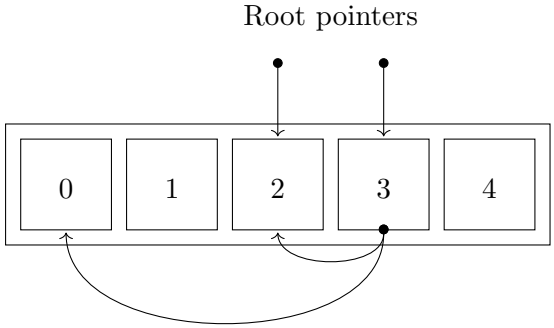


Figure 2.5: Heap objects referenced by pointers. Each pointer corresponds to a work stealing task.

2.6.2 G1 Work Stealing Implementation

The work stealing implementation of G1 [25] is based on the non-blocking work stealing algorithm by Arora, Blumofe and Plaxton [4] (ABP).

The workpile for a thread consists of two data structures, as illustrated in figure 2.6. One of those data structures is a public double ended queue (deque), as proposed by the ABP algorithm [4]. Threads push or pop tasks to or from the bottom (local end) of their own deques. Deques that run out of tasks will cause their owning threads to become thieves. Such thieves will randomly select two other threads' deques, compare their size and select the thread with a larger deque as victim. The thief will then pop a task from the top (global end) of the previously selected victim's deque. Threads try to keep their deques filled as effectively as possible to maximize the amount of successful steals.

The second data structure every thread exclusively owns is an overflow stack. This is a contrast to the original ABP algorithm which allows to resize the deques. Should the thread's public deque in the G1 implementation become full, new tasks will be pushed on the thread's overflow stack instead of the deque. This is because resizing a deque might introduce a significant delay and adds complexity. Another reason to prefer an overflow stack over resizing is that only the owning thread is allowed to operate on the overflow stack. Which makes accesses to the overflow stack faster as it is not necessary to employ any memory barriers when operating on it. Threads will attempt to keep the public deque filled with contents of the private overflow stack, so other threads are able to steal tasks if necessary.

The different operations shown in figure 2.6 (pop, push, steal) illustrate which possibilities a thread has to access it's data structures. The events happening in figure 2.6 could be the following:

- Thread T_0 pops tasks from its overflow stack and pushes them to the deque to keep it filled for possible thieves.
- The deque of T_1 is full, so T_1 pushes the newly generated tasks onto the overflow stack.
- Thread T_2 ran out of tasks and has to steal one from another thread.
- The deque of T_3 is almost full. Depending on the configured threshold it might either access its deque or overflow stack. In this example it chose to pop a task from the deque.

Double Ended Queues

The double ended queues (deques) in the G1 garbage collector are based on an array with pointers to `top` and `bottom`. Where `top` corresponds to head and `bottom` to tail of a typical queue. One major difference between the ABP algorithm [4] and the implementation in G1 is that G1 implements the public deque as ring buffer to not require special handling when top exceeds the length of the backing array.

Since the index `top` may be modified by more than one thread, modification requires a compare-and-swap instruction. The implementation avoids the ABA problem by using a `tag` field. In order to perform CAS on `top` and `tag` at the same time, they are both wrapped into a data structure called `age`. Listing 2.10 shows how a union provides simultaneous access to the combination of `top` and `tag` via

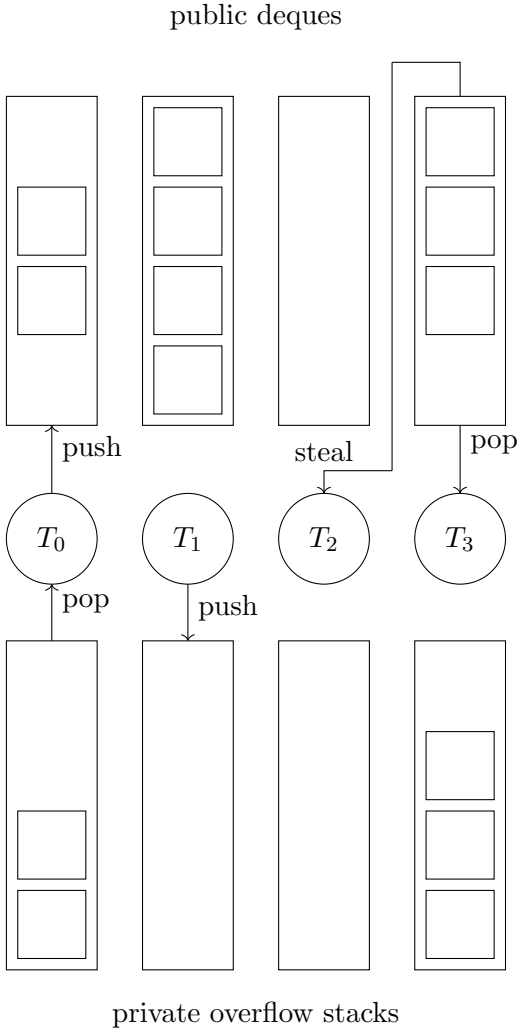


Figure 2.6: Threads and their corresponding data structures.

its member `_data`. Whereas the contained struct allows individual access to `top` and `tag` by accessing the contents of the union's `_fields` member.

```
struct fields {
  uint32_t _top;
  uint32_t _tag;
};
union {
  uint64_t _data;
  fields _fields;
};
```

Listing 2.10: Allows to read or write `_top` and `_tag` simultaneously.

The following list shows the methods of the ABP deque implementation. Their source code can be found in appendix A.

push()
: Adds a task element to the deque. Gets only invoked by the deque's owner. A `release_store()` barrier is needed to ensure that the element is appended to

the queue before the update to the `bottom` index (which refers to the added element) happens. Should the deque's size reach a certain limit, the method `push_slow()` will be invoked. A full deque will result in the element being pushed onto the overflow stack. The code for `push()` is shown in listing A.1.

push_slow

A deque of size $N-1$ indicates that the deque is empty. Therefore, an element can still be pushed onto the queue by copying the element into the underlying array before incrementing the bottom index. The order is ensured via a `release_store()` memory barrier. Details can be found in listing A.2.

pop_local()

As presented in listing A.3, `pop_local()` decrements the `bottom` index and takes an element from the deque. Only the thread owning the deque will invoke this method. Should the element to be taken be the last one in this deque, `pop_local_slow()` is called to resolve potential issues if a thief tries to steal the same element.

pop_local_slow()

When a thief took the last element and incremented `top` while the owning thread decremented `bottom`, the indices will wrap around, making `top` greater than `bottom`. This state needs to be fixed by resetting `top` to the same value as `bottom`. However, if the owning thread took the last element instead of a thief, `top` and `tag` get updated via a CAS instruction to avoid thieves interfering. The details are laid out in listing A.4.

pop_global()

Gets invoked by thieves to steal elements from the deque's top. Listing 2.11 shows the code of this method. To determine if there are elements in the deque, `pop_global()` needs at least a `load_acquire()` memory barrier in line 9. Whereas some CPU architectures even require a full barrier (fence), as seen in line 7. Lines 9 to 12 determine if the deque is empty. Should the deque contain an element though, it gets copied (line 14) before the `top` index and `tag` are updated via a CAS instruction in lines 15 to 17. The success of the `pop_global()` operation depends on this CAS instruction. Should it fail, no element will be taken from the queue, resulting in a failed steal attempt.

Listing 2.11 also serves as an example of how many memory barriers are used in the original G1 work stealing implementation. Two memory barriers are utilized within `pop_global()`'s critical path. Even more important, because of their invocation frequency, are the barriers in `push()` and `pop_local()`. Both methods have one barrier that is required during every push or pop of a task.

```
pop_global(volatile E& t) {
  Age oldAge = _age.get();
  // Architectures with weak memory model require a barrier here
  // to guarantee that bottom is not older than age,
  // which is crucial for the correctness of the algorithm.
#if !(defined SPARC || defined IA32 || defined AMD64)
  OrderAccess::fence();
#endif
  uint localBot = OrderAccess::load_acquire(&_bottom);
  uint n_elems = size(localBot, oldAge.top());
  if (n_elems == 0) {
    return false;
  }
  t = _elems[oldAge.top()];
  Age newAge(oldAge);
  newAge.increment();
  Age resAge = _age.cmpxchg(newAge, oldAge);
  return resAge == oldAge;
}
```

Listing 2.11: The ABP implementation's `pop_global()` method.

Termination Algorithm

After all heap objects have been evacuated and no thread has tasks anymore, the evacuation needs to be terminated properly. The current G1 implementation is doing this via a modified version of the *Optimized Work Stealing Threads (OWST)* [11] task termination algorithm described in listings 2.12 and B.1.

```
bool offer_termination(TerminatorTerminator* terminator) {
  // Single worker, done
  if (_n_threads == 1) {
    _offered_termination = 1;
    return true;
  }

  _blocker->lock_without_safepoint_check();
  _offered_termination++;
  // All arrived, done
  if (_offered_termination == _n_threads) {
    _blocker->notify_all();
    _blocker->unlock();
    return true;
  }

  Thread* the_thread = Thread::current();
  while (true) {
    if (_spin_master == NULL) {
      _spin_master = the_thread;

      _blocker->unlock();

      if (do_spin_master_work(terminator)) {
        return true;
      } else {
        _blocker->lock_without_safepoint_check();
```

```
28          // There is possibility that termination is reached between
        dropping the lock before returning from do_spin_master_work() and
        acquiring lock above.
29          if (_offered_termination == _n_threads) {
30            _blocker->unlock();
31            return true;
32          }
33        }
34      } else {
35        _blocker->wait(true, WorkStealingSleepMillis);
36
37        if (_offered_termination == _n_threads) {
38          _blocker->unlock();
39          return true;
40        }
41      }
42
43      size_t tasks = tasks_in_queue_set();
44      if (exit_termination(tasks, terminator)) {
45        _offered_termination--;
46        _blocker->unlock();
47        return false;
48      }
49    }
50  }
```

Listing 2.12: Termination algorithm for threads that ran out of work [25].

Listing 2.12 shows the termination method every thread calls after they ran out of tasks and failed to steal one from their peers. Basically, every thread marks itself as done by incrementing the `_offered_terminaiton` counter. If the counter is equal to the number of worker threads, it means they have all finished their work and can therefore leave the evacuation phase.

If there exists more than one thread, lines 8 to 14 increment the counter to indicate that this thread is done and check if the other threads have finished as well. If so, the other waiting threads are woken up and the evacuation phase gets terminated. Otherwise, an infinite loop starts at line 18, in which the threads wait until every other thread is done and they all may terminate. But first, in lines 19 to 22 a spin master is chosen. Depending on the spin master's result, it will either return right away or check if the other threads are done as well before returning. In lines 35 to 39, all tasks that did not become the spin master will wait until being woken up, or a certain amount of time has passed. Then they check if termination is possible. Should that not be the case, lines 43 to 47 check if it is required to cancel the termination by checking if there are still threads with tasks that could be stolen.

The spin master's description is listed in appendix B.

Chapter 3

Implementation

This section explains how the algorithms from the idempotent work stealing [22] and task-pushing [44] papers have been implemented.

3.1 Idempotent Work Stealing

Figure 2.4 shows how the G1 evacuation algorithm behaves if an attempt to process a task more than once is made. This behavior of detecting an already evacuated object is necessary for correctness. Multiple references to the same object would otherwise result in duplicated objects on the heap. Figure 2.4 shows how C and D got evacuated. Then D spawned a task to evacuate C again. However, this task gets canceled as soon as it is clear that the object C has already been evacuated. This property makes the evacuation process idempotent, as an operation is idempotent if its repeated application does not change the result after the initial execution. According to Michael et al., such a behavior can be used to relax the semantics during work stealing. They propose to "extract tasks at least once – instead of exactly once" [22], from a thread's deque. In contrast to the current ABP implementation, which guarantees that every task gets removed from a deque exactly once. However, by changing this to allow tasks to get removed multiple times, it can reduce the amount of memory barriers needed in the work stealing algorithm. But also potentially increases the amount of tasks being executed.

Michael et al. present three different approaches to idempotent work stealing [22]:

Idempotent LIFO extraction
: Similar to a stack, every task is extracted and added from one side only. In this case, the tail.

Idempotent FIFO extraction
: All extractions take tasks from the head of the queue. Adding tasks to the queue happens via its tail.

Idempotent double-ended extraction
: The owning thread adds and removes elements from the queue's tail while thieves steal from head.

Due to the order of how tasks are taken from the deques, we initially assumed idempotent double-ended extraction to profit a lot from spatial locality. Thus, we implemented it thinking it would have the best performance under evacuation workloads.

When implementing the FIFO extraction we found that its breadth-first search characteristics accumulated too many elements for the threads' data structures, which is bad for spatial locality of the objects in the heap and reducing mutator performance. Since Michael et al. found LIFO extraction to be the best algorithm in terms of runtime [22], we implemented it as well.

The idempotent work stealing queue implementations are drop-in replacements for the existing public deques. The overflow stack, task terminator and most of the logic for thread interaction, during work stealing, could be taken from the current ABP implementation.

3.1.1 Idempotent Double-Ended Extraction

This section explains how the idempotent double-ended extraction algorithm from Michael et al. [22] works as implemented into G1.

From a high level perspective, the data structures of idempotent double-ended extraction are similar to the ones of the ABP implementation in figure 2.6. Michael et al. suggested to resize the deques if they run out of space. However, we decided to keep the overflow stacks, as resizing a deque might introduce a significant delay and adds complexity. Only the deques got modified to allow tasks being taken more than once, which can result in task duplication during steals. This can happen when thread T_3 takes an element from its own tail by creating a local copy of the head pointer and the size variable. Then it copies the task, but gets preempted before it can update head and size. Now thread T_2 steals a task from the head of the deque. It updates the head pointer as well as the size variable. If T_3 resumes now, it will update the head with a stale value and basically reset it to what it was before T_3 stole the task. If any thread now attempts to steal from T_3, it will steal the same task that was previously stolen by thread T_2.

The deque for double-ended extraction is composed of an array and a variable to store the array's capacity. A pointer to the deque's head, the current size and a tag to mitigate the ABA problem are stored inside the so called `Anchor` data structure. To either access all anchor variables separately or simultaneously, they are wrapped in a structure which is part of a union, as shown in listing 3.1.

```
1  class Anchor {
2    struct Fields {
3      uint32_t head;
4      uint16_t size;
5      uint16_t tag;
6    };
7    union {
8      uint64_t data;
9      Fields fields;
10   };
11 };
```

Listing 3.1: Anchor class. Initializers, accessor methods and mutator methods omitted.

Listing 3.2 shows the implementation of the `put()` method. It first makes a local copy of the anchor data. Then, in line 3 and 4, it checks if the deque is full and returns `false` if so. The caller makes sure the element gets pushed onto the private overflow stack instead. In line 6, a task gets inserted at the end of the deque. This operation

must happen before incrementing the size of the deque. The memory barrier in line 7 guarantees this. Finally, the local `size` and `tag` variables are incremented and written to the globally visible `anchor`.

```
1  bool put(E const task) {
2    Anchor local(anchor.data);
3    if (local.size() >= tasks.size) {
4      return false;
5    }
6    tasks.array[(local.head() + local.size()) % tasks.size] = task;
7    OrderAccess::storestore();
8    local.set_size(local.size() + 1);
9    local.set_tag(local.tag() + 1);
10   anchor.data = local.data;
11   return true;
12 }
```

Listing 3.2: The put method of idempotent double-ended extraction.

After copying the global `anchor` to a local structure in listing 3.3, line 2, the `take()` method checks if the deque is empty or beneath a certain threshold, as shown in line 3. Next, in lines 6 and 7, it copies the element from the deque's tail. Finally, the size is updated and written to the shared anchor structure.

```
1  bool take(E& task, uint threshold) {
2    Anchor local(anchor.data);
3    if (local.size() <= threshold) {
4      return false;
5    }
6    task = tasks.array[(local.head()
7      + local.size() - 1) % tasks.size];
8    local.set_size(local.size() - 1);
9    anchor.data = local.data;
10   return true;
11 }
```

Listing 3.3: The take method of idempotent double-ended extraction.

Stealing begins with the thief selecting a victim, as described in section 2.6.2. Next, the thief calls the victim's `steal()` method, which is shown in listing 3.4. There, in lines 5 and 6, the victim's anchor gets copied and checked if the deque is empty. Line 9 copies a task from the head of the victim's deque. In lines 10 to 14, the new values for the deque's head and size are calculated and stored in an `anchor` variable. Finally, if the `head`, `size` and `tag` values are still the same as when they were read in line 5, they will be atomically exchanged with the new values from lines 10 to 14. In contrast to the idempotent work stealing paper we only need one memory barrier inside the `steal()` method. The second barrier would only be necessary in case a deque gets resized.

```
1  bool steal(E& task) {
2    Anchor local;
3    Anchor exchange_value;
4    do {
5      local = Anchor(anchor.data);
```

```
6       if (local.size() == 0) {
7         return false;
8       }
9       task = tasks.array[local.head() % tasks.size];
10      exchange_value = Anchor(
11          (local.head() + 1) % tasks.size,
12          local.size() - 1,
13          local.tag()
14      );
15    } while (Atomic::cmpxchg(
16        exchange_value.data,
17        &anchor.data,
18        local.data) != local.data);
19    return true;
20  }
```

Listing 3.4: The steal method of idempotent double-ended extraction.

The code presented in this section shows that the `steal()` method needs less memory barriers within its critical path than the `steal()` from the ABP implementation. But most importantly, the idempotent work stealing implementation requires no memory barriers within the frequently called `take()` method.

3.2 Task-pushing

Task-pushing [44] is a work distribution technique that does not rely on taking or stealing tasks from other threads like in traditional work stealing. On the contrary, threads deliberately give tasks to other threads, similar to message passing. Although, task-pushing was initially developed to distribute work during the parallel marking process, the results presented in the paper seem promising for evacuation as well.

Fundamentally, every thread stores its tasks in an exclusively owned queue. The main idea, however, is that threads with a task surplus give tasks to other threads that have run out of work. This happens via *single-writer-single-reader (SWSR)* queues, henceforth called *channels*. Since a channel can only provide one way communication, two channels are needed between every pair of threads. As a result, all threads are connected to each other through a network of channels, as shown in figure 3.1. The total number of channels, if N denotes the number of threads, is $N(N-1)$. Where every thread has $N-1$ output and $N-1$ input channels.

Instead of just exchanging the deques like for idempotent work stealing, it was necessary to implement channels, the communication between them, private deques, and a new task terminator.

3.2.1 Channels (Single-Writer-Single-Reader Queues)

This thesis' implementation of the single-writer-single-reader queue, also known as channel, is a simple queue design which stores its elements in an array where the first and last elements are referred to by a head and tail pointer respectively. Empty or invalid entries in a channel are assigned the `NULL` value. The whole communication network is composed of channels being bundled together into a flattened array representation. As a result thread `i` can access the elements supplied by thread `j` by calling `channels(j,i)`. This makes the channel `<j,i>` the input channel of thread `i`,

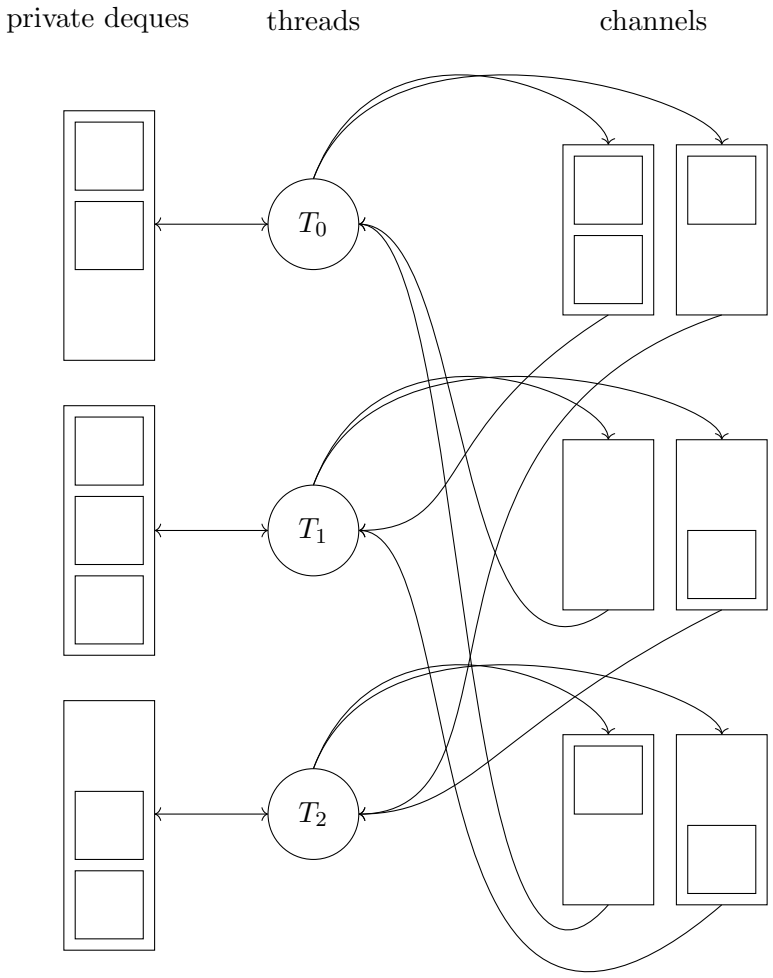


Figure 3.1: Interaction between deques, threads and channels [44].

and an output channel for thread `j`. Similarly, thread `i` can loop over channels `<i,*>` to access each of its output channels.

The major advantage of the channels is that they do not need "atomic CAS operations" [44], given memory accesses are word aligned. Additionally, no memory barriers are required. Stale indices to invalid elements are taken care of by checking every element for `NULL` before processing it. Should an already processed element be taken, it would not be a problem due to the idempotence of evacuation. Listings 3.5 and 3.6 show the gist of how the enqueue and dequeue methods are implemented. The paper of Wu and Li [44] contains a proof of correctness for their single-writer-single-reader queue design.

The length of the channels plays an important role for task distribution. Shorter channels lead to a better task distribution, whereas longer channels have a buffering effect which can increase scalability [44].

```
1  bool enqueue(void* data) {
2    uint tail = _tail;
3    void* old = _entries[tail];
4    if (old != NULL) {
5      return false;
6    }
7    _entries[tail] = data;
8    _tail = (tail + 1) % N;
9    return true;
10 }
```

Listing 3.5: A channel's enqueue method.

```
1  bool dequeue(void*& data) {
2    uint head = _head;
3    data = _entries[head];
4    if (data == NULL) {
5      return false;
6    }
7    _entries[head] = NULL;
8    _head = (head + 1) % N;
9    return true;
10 }
```

Listing 3.6: A channel's dequeue method.

3.2.2 Task Selection Strategy

The main criteria for distributing tasks among threads is availability. Every thread should have a task to work on at any time. Therefore, given that their private deque is not empty, all threads strive to fill their output channels as soon as they notice an empty slot. There exist three main strategies which describe when tasks should be pushed to other threads.

New-task assigning
When the execution of a task spawns a new task, the executing thread will try to put it into the next free output channel it can find.

Old-task dripping
When a thread adds a newly generated task to its dequeue, it also takes a task from the other side of the deque and tries to enqueue it to the next vacant output channel it encounters.

Hybrid task sharing
Recently spawned tasks get passed to other threads. However, if the currently executed task does not generate a new task, one from the private deque is pushed to the next free output channel.

We implemented new-task assigning as well as old-task dripping, and like Wu and Li [44], we found old-task dripping to be the best performing strategy. Hence, we decided to explore it further in this thesis.

3.2.3 Termination Algorithm

Terminating the evacuation phase for task-pushing is more complicated compared to common work stealing techniques, as threads are not able to tell by themselves if they have finished processing all available tasks. Even if a thread has no more tasks to work on, it could get some through its input channels at any time. Hence, it is not able to decide the right moment of termination by itself, but has to wait until all threads are done processing their tasks. As a consequence, one thread, called the spin master, is chosen to supervise the whole termination process. It ensures that every thread's private deque as well as its channels are empty before termination.

Basically, all finished threads (except for the spin master) signal that they are done with their tasks, and wait for the spin master to give them the termination signal. They can abort the termination anytime, by setting `_terminating` to `false`, if necessary. When the spin master receives the information that all other threads are finished, it checks if their channels are still empty. If so, and no other thread canceled the termination, the spin master signals all threads to terminate via `_exit_evacuation`. This termination algorithm was inspired by Peterson's algorithm [33]. Listings 3.7 and 3.8 show the implementation for this thesis.

```
bool offer_termination(TerminatorTerminator* terminator, uint
    worker_id) {
  _blocker->lock_without_safepoint_check();
  Thread* the_thread = Thread::current();
  if (_spin_master == NULL) {
    _spin_master = the_thread;
    _blocker->notify_all();
    _blocker->unlock();
    bool res = do_spin_master_work(terminator, worker_id);
    _blocker->lock_without_safepoint_check();
    _spin_master = NULL;
    _blocker->notify_all();
    _blocker->unlock();
    return res;
  }
  _blocker->notify_all();
  _blocker->unlock();

  q_set = _queue_set;
  _has_work[worker_id] = true;

  if (q_set->queue(worker_id)->has_elements()) {
    _terminating = false;
    return false;
  }
  for (uint i = 0; i < _n_threads; i++) {
    if (q_set->channels(i, worker_id)->has_elements()) {
      _terminating = false;
      return false;
    }
  }
  _has_work[worker_id] = false;
  for (;;) {
    for (uint i = 0; i < _n_threads; i++) {
      if (q_set->channels(i, worker_id)->has_elements()) {
        _terminating = false;
        _has_work[worker_id] = true;
        return false;
      }
    }
    OrderAccess::storeload();
    if (_exit_evacuation) {
      return true;
    }
  }
}
```

Listing 3.7: Termination algorithm for threads that ran out of work.

Listing 3.7 shows the implementation of the termination algorithm for non-spin master threads. In lines 2 to 16 the spin master gets chosen. The first thread entering this section will become the spin master, if no other thread is currently having this role. The array in line 19 holds a boolean value, representing every thread's status. With this, the other threads can signal the spin master that they have finished all their tasks. Before checking its own private queue and input channels, the executing thread marks itself as still having tasks to work on. Then in lines 21 to 30, the thread makes sure its private deque and input channels are empty. Otherwise, it would block the termination, by setting `_terminating` to `false`, and return to work on those newly discovered tasks. If there are no tasks left though, the thread will mark itself as being done and ready for termination (line 31). Next, lines 32 to 44 show, that it will stay in a loop and continuously check if either a new task arrives, or the spin master gives the signal to terminate via the `_exit_evacuation` flag.

```
1  bool do_spin_master_work(TerminatorTerminator* terminator, uint
       worker_id) {
2    q_set = _queue_set;
3    _terminating = false;
4    _has_work[worker_id] = true;
5
6    if (q_set->queue(worker_id)->has_elements()) {
7      return false;
8    }
9    for (uint i = 0; i < _n_threads; i++) {
10     if (q_set->channels(i, worker_id)->has_elements()) {
11       return false;
12     }
13   }
14   _terminating = true;
15   OrderAccess::storeload();
16   for (uint i = 0; i < _n_threads; i++) {
17     if (i != worker_id && _has_work[i]) {
18       return false;
19     }
20     for (uint j = 0; j < _n_threads; j++) {
21       if (q_set->channels(i, j)->has_elements()) {
22         return false;
23       }
24     }
25   }
26   if (_terminating) {
27     _exit_evacuation = true;
28     return true;
29   }
30   return false;
31 }
```

Listing 3.8: Termination algorithm for the spin master thread.

The implementation of the spin master algorithm is shown in Listing 3.8. In lines 6 to 13, the spin master thread makes sure its own private deque and its input channels are empty. Then, in lines 16 to 25, it checks if all other threads have marked themselves as being done. Additionally, the spin master verifies if the other threads' channels are empty. If that is the case and no other thread aborted the termination sequence, the spin master will give the signal to exit via the `_exit_evacuation` flag.

Chapter 4

Experimental Setup

In this chapter the hard- and software setup as well as the benchmarks used for testing the implementations presented earlier, are shown. Additionally it describes which measurements were taken from said benchmarks.

4.1 Hardware

The test machine's CPU was an Intel® Core™ i7-4702HQ [14] processor with 4 physical cores and a base frequency of 2.20 GHz per core. To avoid frequency scaling and archive more consistent results, the CPU's frequency got pinned to its base frequency. The upper limit was set by disabling Intel® Turbo Boost via the command `echo "1" | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo`. Whereas the lower limit got fixed by enabling the performance governor with `sudo cpufreq-set --governor performance`.

Main memory consisted of two 8 GB DDR3 modules with a clock speed of 1600 MHz, resulting in a total capacity of 16 GB of RAM.

Benchmark logs were written to a RAM disk for subsequent extraction of garbage collection pause times.

4.2 Software

The operating system for benchmarking was Ubuntu 20.04.1 LTS (Focal Fossa) [42] with Linux kernel 5.4.0-47-generic.

4.2.1 Development Tools

OpenJDK version 12u [25] was used to implement and test the work stealing alternatives. This version was mostly selected because it has a stable code base and no serious modifications from the maintainers were to be expected. Since OpenJDK is developed in C++, and the OpenJDK building documentation suggests to use the *GNU Compiler Collection, GCC 9.3.0* [9] was used for compilation. Additionally, the *Apache NetBeans* IDE 11 [3] was used for development, as the OpenJDK repository already provides NetBeans project files.

4.3 Measurements

Since the goal of this thesis was to reduce G1's garbage collection pause times, mainly pause time measurements, and in some cases the steal frequencies and termination attempts, were taken during the benchmarks. As suggested by Jain [16], the median of all pause times for each benchmark was calculated to get the final measurement. This was then used to compare the different work stealing implementations to each other.

For better comparability and stressing of the evacuation phase we chose heap and generation sizes such that a significant amount of time has been spent there.

4.4 Benchmark Setup

We used the benchmark suits SPECjvm 2008 [40] (version 1.01), DaCapo [5] (version 9.12-bach-MR1), DaCapo Scala [38] (version 0.1.0-20120216.103539-3), SPECjbb 2005 [39] and pjbb [6].

We initially performed an evaluation to find which benchmarks even benefit from work stealing. Since workloads might already be relatively well distributed over the existing cores, not needing any significant amount of work stealing to distribute their workload. Therefore, every benchmark was run once with the ABP algorithm having work stealing enabled, and once with the ABP algorithm having work stealing disabled. We disabled work stealing by changing the `steal()` method to always return `false`. Table 4.2 lists the comparison of garbage collection pause times of work stealing enabled versus disabled and shows that for several benchmarks work stealing does not improve them. Based on this first comparison in table 4.2 we selected a final set of benchmarks to evaluate the idempotent work stealing and task-pushing implementations with. Table 4.1 shows the selected set of benchmarks and their arguments.

To increase GC pause times, the largest possible workload was selected for every benchmark.

The following list shows all JVM arguments that were used during the benchmarks:

`-Xmn{size}`
: Initial and maximum size of the eden space. Like `Xms` and `Xms` it was used to set the heap and generation sizes such that a significant amount of time has been spent inside the evacuation phase.

`-Xms{size}`
: Minimum heap size.

`-Xmx{size}`
: Maximum heap size.

`-XX:+DisableExplicitGC`
: Ignores the invocation of `System.gc()`, which some of the benchmarks use before they start.

`-XX:ParallelGCThreads=4`
: Defines the number of threads that work on garbage collection. As some preliminary benchmark runs showed that the GC process does not benefit from

virtual cores, it was set equal to the number of physical cores of the test machine.

`-XX:-UseDynamicNumberOfGCThreads`
Instead of using a dynamic number of threads for each garbage collection, the maximum amount (as defined in `-XX:ParallelGCThreads`) was selected to decrease variability of the runs.

`-XX:+AlwaysPreTouch`
Makes sure every page of the heap is mapped into the main memory during initialization of the virtual machine. This helps to increase consistency between benchmark runs.

`-Xlog:gc=debug,gc+stats=debug,gc+heap=debug,gc+phases=debug:{file}`
Enables JVM logging. Writes all log messages tagged with gc, heap, phases, and stats at a *debug* level, to the given file.

	Benchmark	Benchmark Arguments	Additional JVM Arguments
DaCapo	h2	-n 10 -s huge	-Xmn2G -Xms6G -Xmx6G
	jython	-n 10 -s large	-Xmn1G -Xms2G -Xmx2G
	pmd	-n 10 -s large	
	tomcat	-n 10 -s huge	-Xmn2G -Xms6G -Xmx6G
	tradebeans	-n 10 -s huge	-Xmn4G -Xms12G -Xmx12G
	xalan	-n 10 -s large	-Xmn1G -Xms3G -Xmx3G
DaCapo Scala	apparat	-n 10 -s gargantuan	-Xmn1G -Xms2G -Xmx2G
	factorie	-n 10 -s gargantuan	-Xmn8G -Xms10G -Xmx10G
	scalac	-n 10 -s large	-Xmn256M -Xms1G -Xmx1G
	scalatest	-n 10 -s huge	-Xmn1G -Xms4G -Xmx4G
	scalaxb	-n 10 -s huge	-Xmn256M -Xms2G -Xmx2G
	tmt	-n 10 -s huge	-Xmn6G -Xms12G -Xmx12G
SPECjvm	scimark.lu.large	--ignoreCheckTest -ikv -i 1	-Xmn512M -Xms3G -Xmx3G
	xml.validation	--ignoreCheckTest -ikv -i 1	

Table 4.1: JVM arguments used during the benchmarks.

	Benchmark	WS Enabled [ms]	WS Disabled [ms]
DaCapo	avrora	9.507	9.369
	batik	7.755	12.087
	eclipse	7.171	7.688
	fop	8.868	11.985
	h2	295.980	296.668
	jython	23.24	54.728
	luindex	8.945	8.902
	lusearch	9.489	11.354
	pmd	13.248	12.226
	tomcat	18.170	19.018
	tradebeans	119.582	194.725
	xalan	17.653	17.058
DaCapo Scala	actors	5.499	10.285
	apparat	28.201	84.916
	factorie	2,129.717	2,150.173
	kiama	7.508	7.901
	scalac	38.022	39.085
	scaladoc	6.848	7.632
	scalap	6.789	11.924
	scalariform	7.648	10.426
	scalatest	16.393	27.425
	scalaxb	13.028	28.807
	tmt	12.275	19.632
pJBB2005	pjbb2005	19.104	19.539
SPECjbb2005	specjbb2005	24.768	26.507
SPECjvm	compress	0.964	0.981
	crypto.aes	1.113	0.949
	crypto.rsa	9.161	10.062
	crypto.signverify	0.986	0.976
	derby	6.709	7.052
	mpegaudio	6.056	8.328
	scimark.fft.large	12.352	11.032
	scimark.fft.small	1.235	1.515
	scimark.lu.large	51.023	52.527
	scimark.lu.small	8.302	8.660
	scimark.sor.large	5.976	8.143
	scimark.sor.small	4.494	8.789
	scimark.sparse.large	8.595	9.878
	scimark.sparse.small	6.580	9.232
	serial	2.865	3.026
	sunflow	6.813	7.243
	xml.transform	4.512	6.062
	xml.validation	42.484	63.276

Table 4.2: Initial comparison of median GC pause times of the ABP algorithm with and without work stealing. Each benchmark was run once.

Chapter 5

Evaluation

This chapter evaluates the benchmark results of the ABP algorithm, as it is currently used within G1, compared to the newly implemented alternatives, idempotent work stealing and task-pushing. We first selected the best strategy yielding the highest performance before comparing them to ABP. Finally, we explain two factors which might have influenced the performance of the alternative implementations.

5.1 Results

Since we implemented more than one version for idempotent work stealing and task-pushing, it was necessary to determine the most performant variant of those algorithms before comparing them to the ABP algorithm.

For idempotent work stealing, two different approaches of how to access the task queues were implemented and benchmarked: idempotent LIFO extraction and idempotent double-ended extraction.

Benchmark runs for the task-pushing algorithm helped to find the channel size which resulted in the lowest GC pause times.

The tables showing the median pause times of all benchmark runs can be found in appendix C.

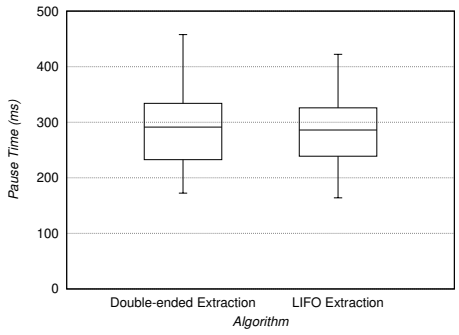
All box plots in figures 5.1, 5.2 and 5.3 have their outliers removed, as Georges et al. [10] suggest to exclude them when evaluating Java performance.

The graphs for the benchmarks *jython*, *tomcat*, *xalan*, *scalaxb* and *tmt* show rather low GC pause times in figures 5.1, 5.2 and 5.3. This is different than it was during the initial runs, shown in 4.2. Those runs were executed with only one iteration and decided the selection of those benchmarks. However, running them with a higher number of iterations lead the graphs to show way lower GC pause times. The early iterations of the benchmarks showed GC pause times so high, compared to the majority of following iterations, that they were considered outliers in the box plots, and are therefore not visible.

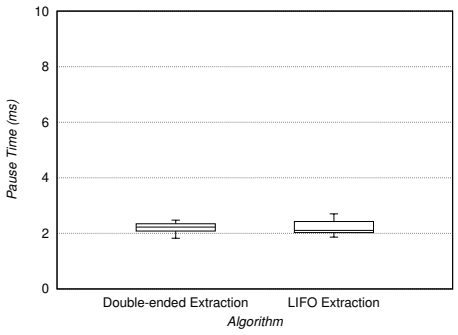
5.1.1 Idempotent Work Stealing Strategy

Each graph in figure 5.1 shows the comparison of garbage collection pause times between the idempotent LIFO extraction and idempotent double-ended extraction implementations. Although none of these implementations were able to decide all of the benchmarks in their favor, the graphs show that the idempotent LIFO extraction strategy slightly prevails in the majority of the benchmarks. This is consistent

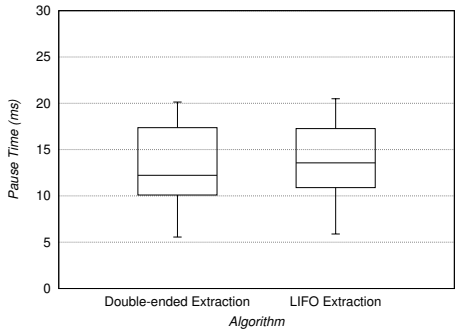
with the findings of Michael et al. in their idempotent work stealing paper [22]. Consequently, it was chosen to compete against the ABP and task-pushing implementations.



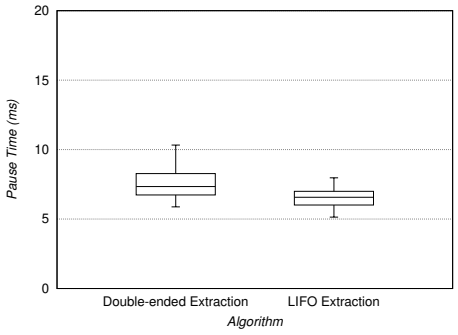
(a) DaCapo h2.



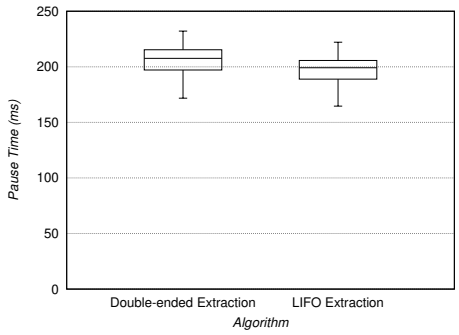
(b) DaCapo jython.



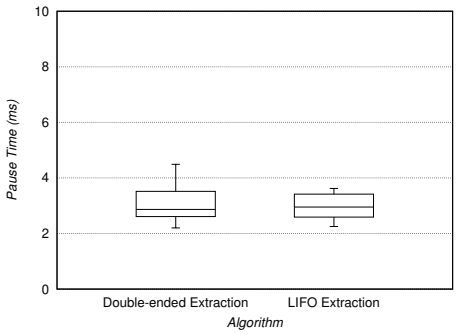
(c) DaCapo pmd.



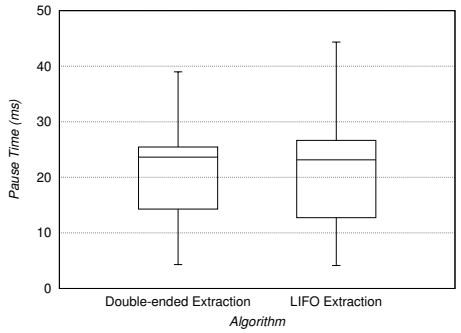
(d) DaCapo tomcat.



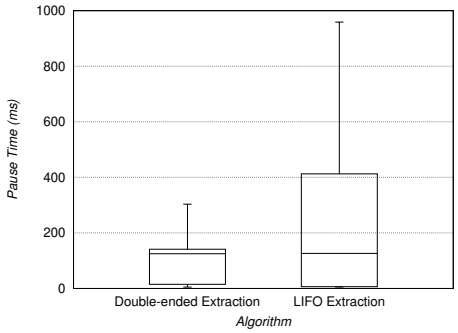
(e) DaCapo tradebeans.



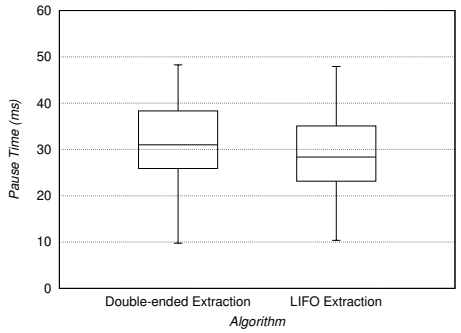
(f) DaCapo xalan.



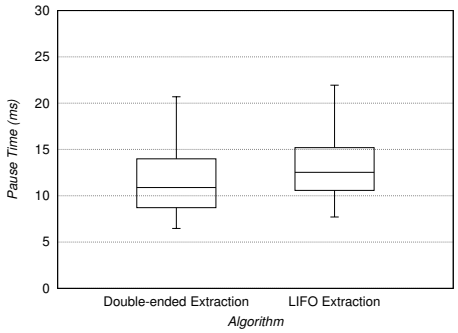
(g) Scala apparat.



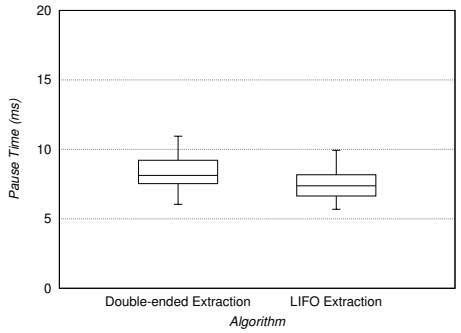
(h) Scala factorie.



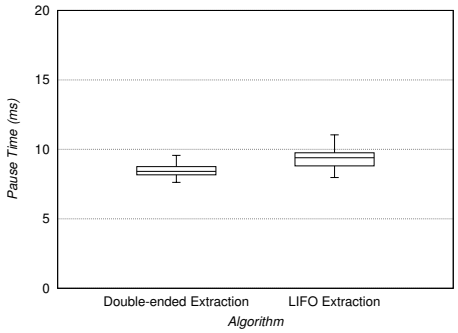
(i) Scala scalac.



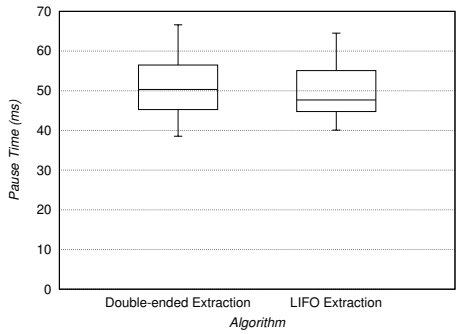
(j) Scala scalatest.



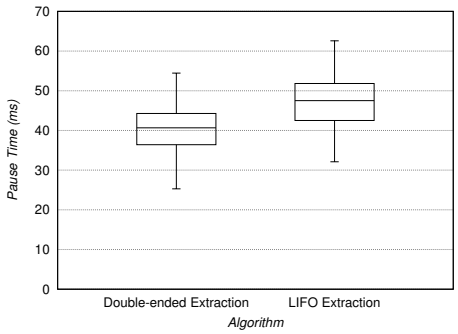
(k) Scala scalaxb.



(l) Scala tmt.



(m) SPECjvm2008 scimark.lu.large

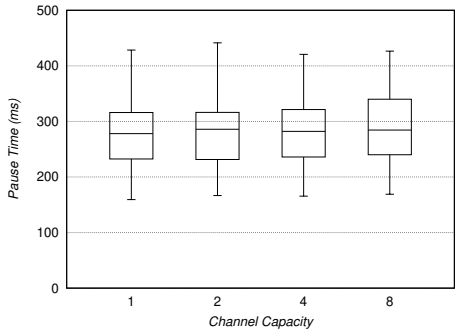


(n) SPECjvm2008 xml.validation

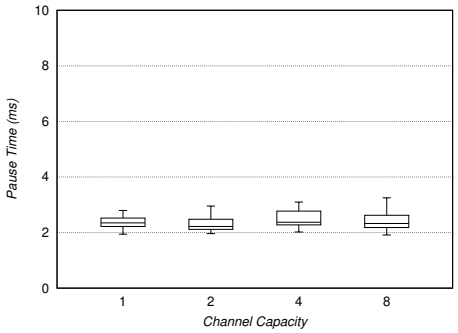
Figure 5.1: Pause time comparison of idempotent Work Stealing algorithms; Double-ended extraction and LIFO extraction.

5.1.2 Task-pushing Strategy

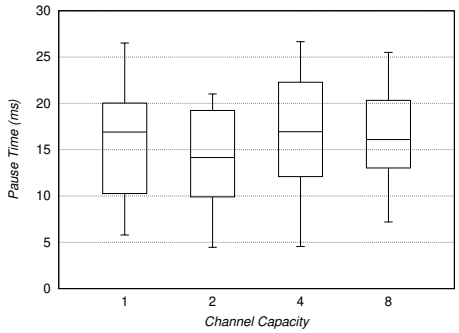
The box plots in figure 5.2 show what impact the channel size can have on garbage collection pause times when using the task-pushing algorithm with the old-task dripping strategy. There is unfortunately not a single channel size that is best for all workloads. However, a channel size of one had the lowest GC pause times during the majority of the benchmarks. Wu and Li similarly found shorter queues/channels to work better than longer ones [44]. Where a channel length of one had the best results for the *pseudojbb* benchmark, while a channel size of two is best suited for *GCOld*.



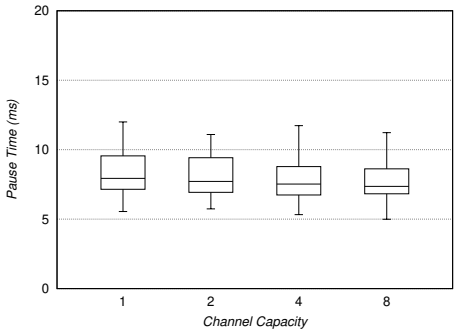
(a) DaCapo h2.



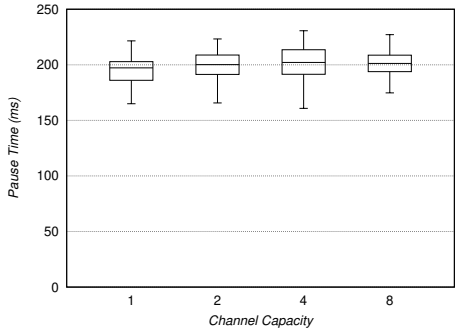
(b) DaCapo jython.



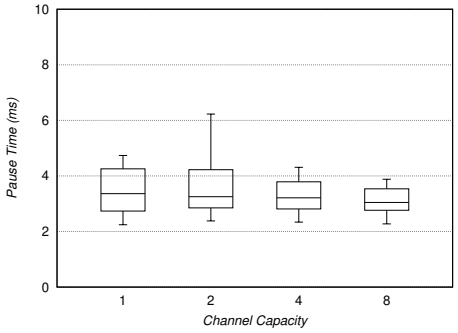
(c) DaCapo pmd.



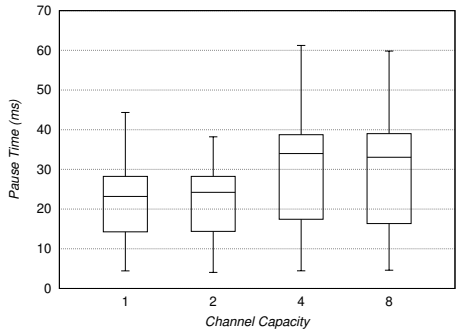
(d) DaCapo tomcat.



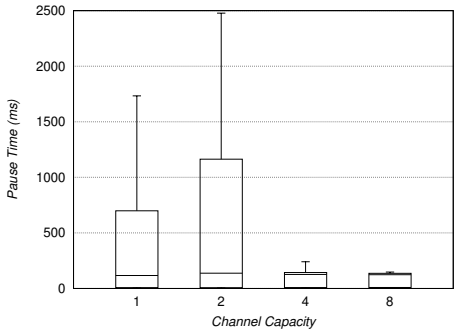
(e) DaCapo tradebeans.



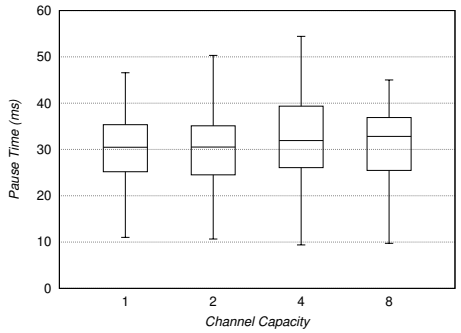
(f) DaCapo xalan.



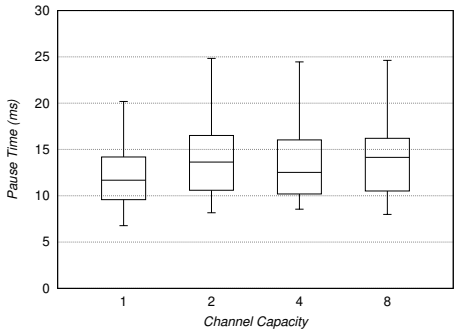
(g) Scala apparat.



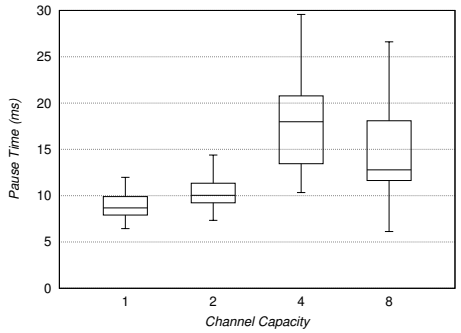
(h) Scala factorie.



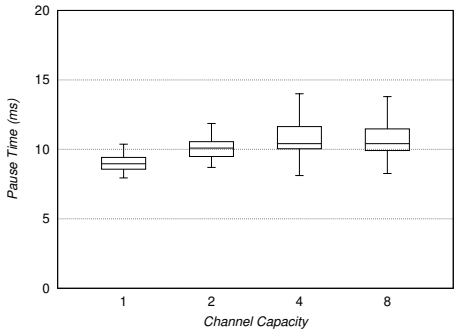
(i) Scala scalac.



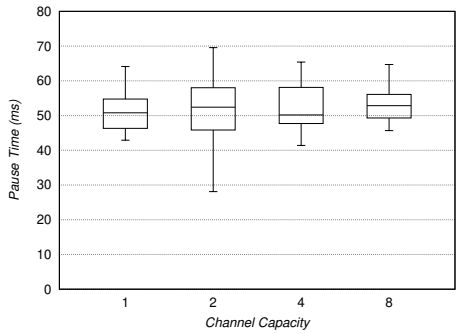
(j) Scala scalatest.



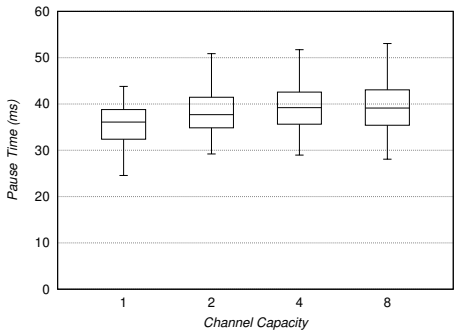
(k) Scala scalaxb.



(l) Scala tmt.



(m) SPECjvm2008 scimark.lu.large.

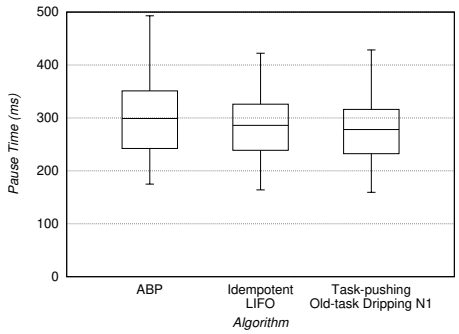


(n) SPECjvm2008 xml.validation.

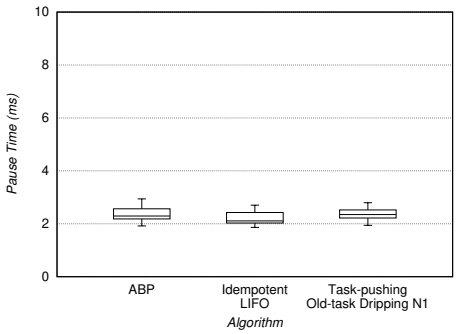
Figure 5.2: Pause time comparison of the Task-pushing algorithm using old-task dripping strategy with different channel sizes.

5.1.3 ABP vs. Idempotent Work Stealing vs. Task-pushing

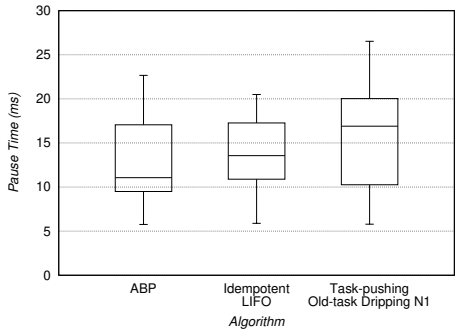
Figure 5.3 shows the pause time comparison of ABP, idempotent work stealing and task-pushing. In seven out of 14 benchmarks, idempotent work stealing (with LIFO extraction) proved to have lower pause times than the other algorithms. Second is task-pushing (old-task dripping with a channel length of one), followed by ABP. Nevertheless, the results are quite close to each other. Even the distribution of the pause times is quite similar among all three algorithms, except for the Scala *factorie* benchmark. Here the third and fourth quartiles of task-pushing show an extremely high spread compared to ABP and idempotent work stealing.



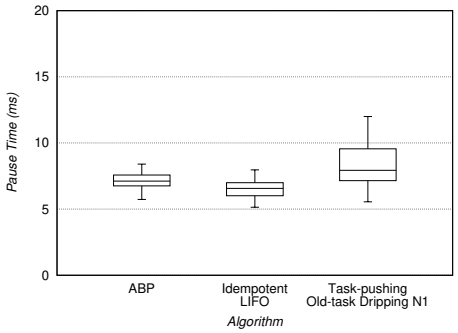
(a) DaCapo h2.



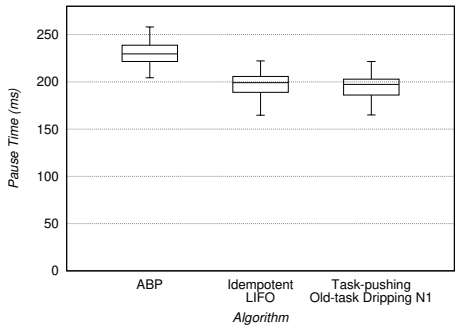
(b) DaCapo jython.



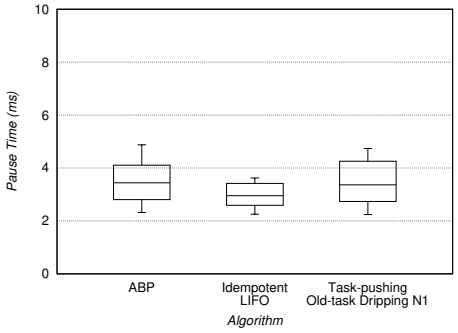
(c) DaCapo pmd.



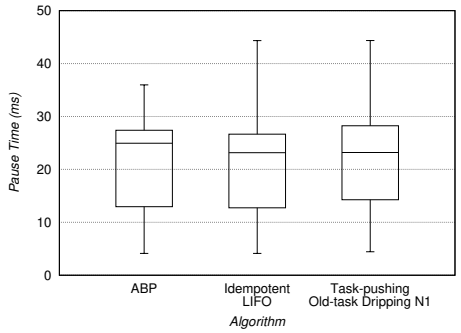
(d) DaCapo tomcat.



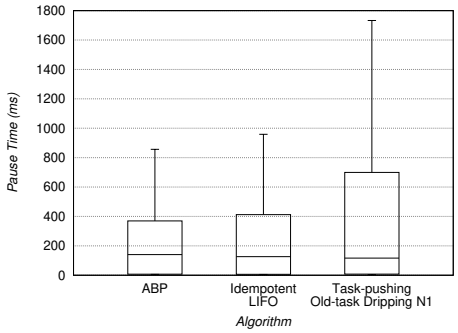
(e) DaCapo tradebeans.



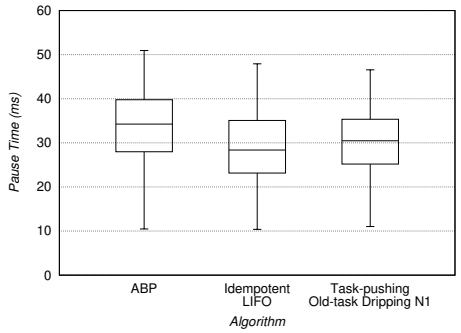
(f) DaCapo xalan.



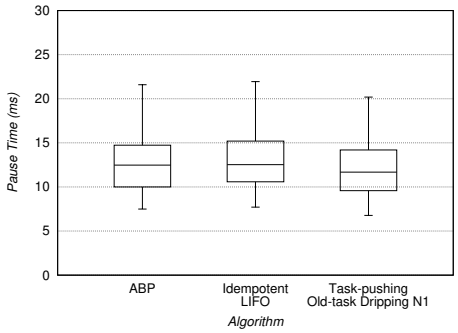
(g) Scala apparat.



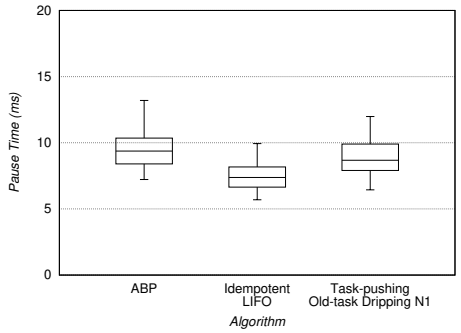
(h) Scala factorie.



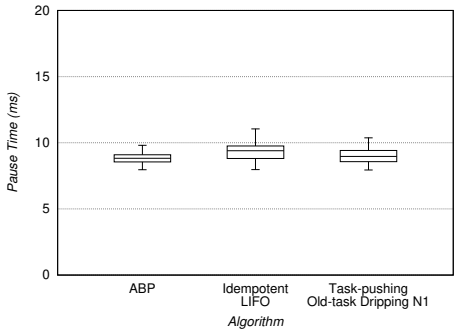
(i) Scala scalac.



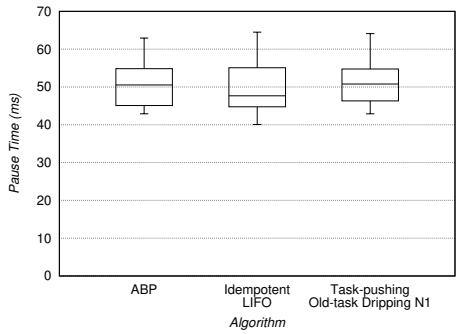
(j) Scala scalatest.



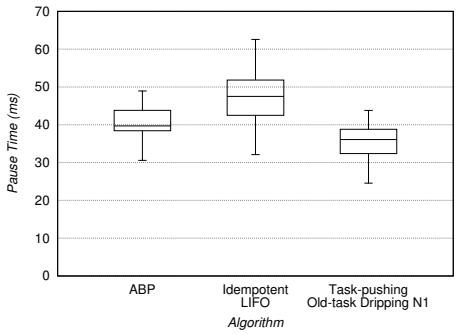
(k) Scala scalaxb.



(l) Scala tmt.



(m) SPECjvm2008 scimark.lu.large.



(n) SPECjvm2008 xml.validation.

Figure 5.3: Pause time comparison of ABP, idempotent work stealing and task-pushing algorithms.

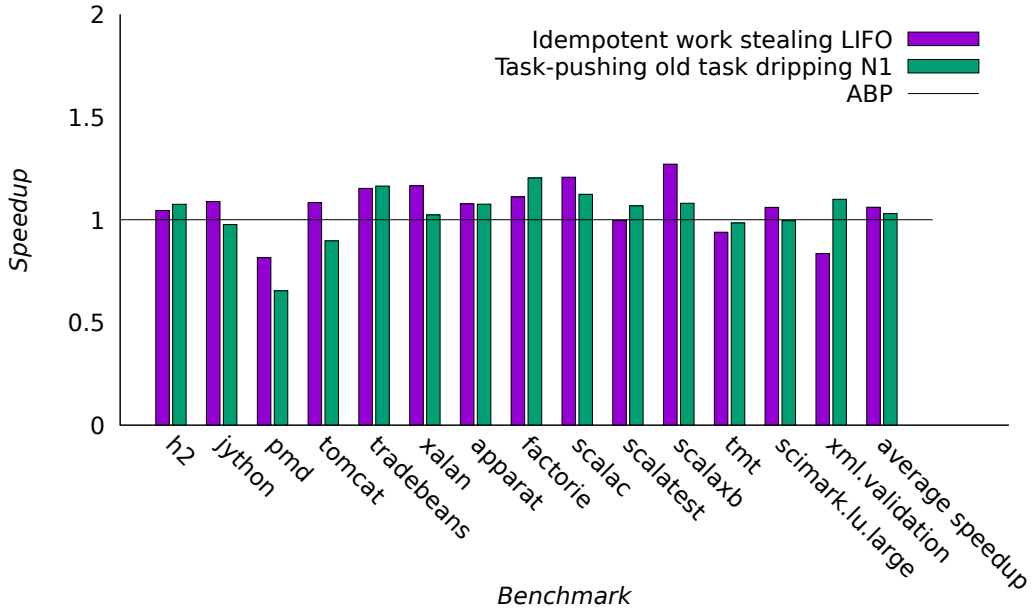


Figure 5.4: Speedup comparison between ABP, idempotent work stealing and task-pushing.

The graph in figure 5.4 and the table 5.1 give an overview of the speedups over all benchmarks of idempotent work stealing and task-pushing compared to ABP. The median garbage collection pause time results of ABP are the baseline and represented as a horizontal line in the graph. It shows that most of the time one of the alternative approaches is faster than ABP. Although, not as consistent and fast as we had initially expected.

5.2 Idempotent Work Stealing and Task-pushing Overhead

This section presents observations of two attributes the alternative implementations have, that might impair their performance.

5.2.1 Task Duplication Through Stealing

The number of steals the threads execute during evacuation might give a hint why idempotent work stealing does not outperform ABP as much as we initially assumed. Due to the relaxed steal semantics, idempotent work stealing steals by far more tasks that ABP does. This means that idempotent work stealing introduces a lot overhead due to duplicated tasks which G1 has to discard when it discovers that those tasks had already been completed. Table 5.2 shows a comparison of the number of successful steals between ABP and idempotent work stealing.

5.2.2 Number of Termination Attempts

A reason why task-pushing did not perform as good as initially expected might have to do with the task termination. The higher termination complexity, compared to ABP and idempotent work stealing, seems to have a negative impact on how reliable

	Benchmark	Speedup factor: Idempotent work stealing LIFO	Speedup factor: Task-pushing old task dripping N1
DaCapo	h2	1.045	1.075
	jython	1.089	0.977
	pmd	0.816	0.654
	tomcat	1.084	0.897
	tradebeans	1.153	1.164
	xalan	1.165	1.023
DaCapo Scala	apparat	1.078	1.076
	factorie	1.111	1.203
	scalac	1.207	1.124
	scalatest	0.995	1.068
	scalaxb	1.270	1.080
	tmt	0.939	0.985
SPECjvm	scimark.lu.large	1.060	0.995
	xml.validation	0.835	1.100

Table 5.1: Speedup comparison between ABP, idempotent work stealing and task-pushing.

	Benchmark	Median number of steals: ABP	Median number of steals: Idempotent work stealing LIFO
DaCapo	h2	120.5	221.0
	jython	1,009.5	7,187.0
	pmd	274.0	2,038.0
	tomcat	8,113.5	4,973.0
	tradebeans	987.0	15,542.0
	xalan	280.0	320.0
DaCapo Scala	apparat	1,513.0	4,012.0
	factorie	5,778.0	6,255.0
	scalac	360.0	1,271.5
	scalatest	13,422.0	10,838.5
	scalaxb	1,024.5	4,363.5
	tmt	734.0	9,581.0
SPECjvm	scimark.lu.large	190.5	1,473.0
	xml.validation	4,312.0	30,552.0

Table 5.2: Median number of successful steals during the evacuation phase.

	Benchmark	Median number of termination attempts		
		ABP	Idempotent work stealing LIFO	Task-pushing old task dripping N1
DaCapo	h2	4.0	4.0	15.5
	jython	4.0	4.0	93.5
	pmd	4.0	4.0	41.0
	tomcat	86.5	10.0	66.0
	tradebeans	4.0	4.0	128.0
	xalan	9.5	15.5	138.0
DaCapo Scala	apparat	5.0	4.0	339.0
	factorie	4.0	4.0	4.0
	scalac	4.0	4.0	90.5
	scalatest	150.0	47.0	754.0
	scalaxb	4.0	4.5	193.5
	tmt	4.0	4.0	149.0
SPECjvm	scimark.lu.large	4.0	4.0	46.0
	xml.validation	4.0	4.0	128.0

Table 5.3: Median number of termination attempts during the evacuation phase.

task-pushing can terminate the evacuation phase. Table 5.3 shows that task-pushing requires significantly more termination attempts, than ABP and idempotent work stealing, to successfully end an evacuation.

Chapter 6

Summary and Outlook

This chapter concludes the thesis and gives an outlook on which details in this area it might be beneficial for future research to focus on.

6.1 Conclusion

The intention of this thesis was to find a replacement for the ABP algorithm which is currently used in the Garbage-First garbage collector of the HotSpot virtual machine. We selected and implemented two very different work stealing algorithms to manage the evacuation phase during garbage collection and reduce its stop-the-world pause times. The algorithms picked for this task were chosen because of their reduction in the amount of memory barriers executed, compared to the ABP implementation. As those memory barriers are quite costly, we assumed that reducing them might result in a performance gain during evacuation. The chosen *idempotent work stealing* and *task-pushing* algorithms promised the following advantages:

Idempotent Work Stealing
: Is able to reduce the number of memory barriers by relaxing the work stealing semantics to allow tasks to be processed more than once. Therefore, it is able to cut down on memory barriers for the cost of work being potentially duplicated.

Task-pushing
: Threads communicate via single-writer-single-reader queues, which allows this algorithm to work without memory barriers for transmitting tasks between its workers. However, this comes at the cost of an increased complexity of the task termination algorithm.

The benchmarks show that our implementations of idempotent work stealing and task-pushing are able to compete with, or sometimes even outperform, the current ABP implementation during the evacuation phase. However, the new implementations could not reduce the overall pause times to a level that would justify replacing the currently used implementation of the well understood and reliable ABP work stealing algorithm. Even more so as there is currently limited data about the new algorithms' behavior in different kinds of workloads.

6.2 Future Work

Future research should consider the potential effects of task duplication more carefully, as it seems to be the major drawback of idempotent work stealing. Reducing

the amount of duplicated work might have a tremendously positive impact on performance.

In this thesis we only implemented idempotent work stealing and task-pushing for the evacuation phase of G1. However, there is other work where G1 currently employs work stealing. Like the concurrent marking phase which uses work stealing to distribute marking work. Hence, it would be interesting to implement idempotent work stealing and task-pushing for the marking phase as well. Even more so as task-pushing was initially developed as a work distribution algorithm for the marking process.

Furthermore, an interesting approach to help reduce communication overhead would be a *steal-multiple* strategy, which can steal more than one task at a time from other threads.

Finally, idempotent work stealing and task-pushing should also be tested on machines with a higher CPU core count, to get an overview of how well they scale when it comes larger CPU core counts.

Bibliography

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 219–228, 2013.

[2] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual: Volumes 1-5*, Apr 2020.

[3] Apache Software Foundation. Apache NetBeans. `https://netbeans.apache.org/`. [Online; Accessed on 02-Feb-2021].

[4] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct 2006. ACM Press.

[6] Steve Blackburn. pjbb2005. `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005/`. [Online; Accessed on 28-Sep-2020].

[7] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, 2010.

[8] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, page 37–48, New York, NY, USA, 2004. Association for Computing Machinery.

[9] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. `https://gcc.gnu.org`, Jan 2021. [Online; Accessed on 02-Feb-2021].

[10] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*,

volume 42 of *OOPSLA '07*, pages 57–76, New York, NY, USA, October 2007. ACM. doi: 10.1145/1297105.1297033.

[11] Wessam Hassanein. Understanding and Improving JVM GC Work Stealing at the Data Center Scale. ISMM 2016, page 46–54, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[13] ARM Holdings. *ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, 2019.

[14] Intel Corporation. Intel® Core™ i7-4702HQ Processor. `https://ark.intel.com/content/www/us/en/ark/products/75118/intel-core-i7-4702hq-processor-6m-cache-up-to-3-20-ghz.html`. [Online; Accessed on 28-Sep-2020].

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, Apr 2020.

[16] R. Jain. *Art of Computer Systems Performance Analysis: Techniques For Experimental Design Measurements Simulation and Modeling*. Wiley, 2015. doi: 10.1145/3133876.

[17] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, 2016.

[18] Clyde P Kruskal, Larry Rudolph, and Marc Snir. Efficient Synchronization of Multiprocessors with Shared Memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4):579–601, 1988.

[19] Doug Lea. The JSR-133 Cookbook for Compiler Writers. `https://mortoray.com/2010/11/18/cpu-reordering-what-is-actually-being-reordered/`, Mar 2011. [Online; Accessed on 04-Oct-2020].

[20] Paul E McKenney. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton*, 2010.

[21] Paul E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2019.12.22a), 2019.

[22] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent Work Stealing. *SIGPLAN Not.*, 44(4):45–54, Feb 2009.

[23] Mark Moir. Practical Implementations of Non-Blocking Synchronization Primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pages 219–228. Association for Computing Machinery, 1997.

[24] Mortoray, Edaqa. CPU Reordering – What is actually being reordered? `https://mortoray.com/2010/11/18/cpu-reordering-what-is-actually-being-reordered/`, Nov 2010. [Online; Accessed on 02-Oct-2020].

[25] OpenJDK. jdk12u. http://hg.openjdk.java.net/jdk-updates/jdk12u/rev/390566f1850a, Jul 2019.

[26] OpenJDK. Memory Access Ordering Model. http://hg.openjdk.java.net/jdk-updates/jdk12u/file/390566f1850a/src/hotspot/share/runtime/orderAccess.hpp, Jul 2019. [Online; Accessed on 04-Oct-2020].

[27] Oracle. The Garbage First Garbage Collector. https://www.oracle.com/java/technologies/javase/hotspot-garbage-collection.html. [Online; Accessed on 09-Aug-2020].

[28] Oracle. JDK 12 Documentation. https://docs.oracle.com/en/java/javase/12/index.html, March 2019. [Online; Accessed on 09-Aug-2020].

[29] Oracle Corporation. *Oracle SPARC Architecture 2011*, Jan 2016.

[30] Oracle Corporation. Java Language and Virtual Machine Specifications. https://docs.oracle.com/javase/specs/index.html, 2020. [Online; Accessed on 21-Jan-2021].

[31] Oracle Corporation. OpenJDK. http://openjdk.java.net/, 2020. [Online; Accessed on 28-Sep-2020].

[32] Parhar, Poonam. Release Note: G1 now collects unreachable Humongous objects during young collections. https://bugs.openjdk.java.net/browse/JDK-8166785, Sep 2016. [Online; Accessed on 15-Apr-2021].

[33] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115 – 116, 1981.

[34] Jeff Preshing. An Introduction to Lock-Free Programming. https://preshing.com/20120612/an-introduction-to-lock-free-programming/, Jun 2012. [Online; Accessed on 15-Apr-2021].

[35] Jeff Preshing. Atomic vs. Non-Atomic Operations. https://preshing.com/20130618/atomic-vs-non-atomic-operations, Jun 2013. [Online; Accessed on 24-Oct-2020].

[36] Junjie Qian, Witawas Srisa-an, Du Li, Hong Jiang, Sharad Seth, and Yaodong Yang. Smartstealing: Analysis and Optimization of Work Stealing in Parallel Garbage Collection for Java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 170–181. 2015.

[37] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, page 237–245, New York, NY, USA, 1991. Association for Computing Machinery.

[38] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM.

[39] Standard Performance Evaluation Corporation. SPECjbb2005 (RETIRED: October 2013). https://www.spec.org/jbb2005, Dec 2017. [Online; Accessed on 28-Sep-2020].

[40] Standard Performance Evaluation Corporation. SPECjvm® 2008. https://www.spec.org/jvm2008, Aug 2020. [Online; Accessed on 28-Sep-2020].

[41] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[42] These release notes for Ubuntu 20.04 LTS. ReleaseNotes. https://wiki.ubuntu.com/FocalFossa/ReleaseNotes, Sep 2020. [Online; Accessed on 28-Sep-2020].

[43] Tom van Dijk and Jaco C van de Pol. Lace: Non-blocking split deque for work-stealing. In *European Conference on Parallel Processing*, pages 206–217. Springer, 2014.

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

[45] Taiichi Yuasa. Real-Time Garbage Collection on General-Purpose Machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

Listings

List of Figures

List of Tables

Appendix A

Original G1 Deque Methods

```
push(E t) {
  uint localBot = _bottom;
  assert(localBot < N, "_bottom out of range.");
  idx_t top = _age.top();
  uint dirty_n_elems = dirty_size(localBot, top);
  assert(dirty_n_elems < N, "n_elems out of range.");
  if (dirty_n_elems < max_elems()) {
    _elems[localBot] = t;
    OrderAccess::release_store(&_bottom, increment_index(localBot));
    return true;
  } else {
    return push_slow(t, dirty_n_elems);
  }
}
```

Listing A.1: The `push()` method of the ABP implementation.

```
push_slow(E t, uint dirty_n_elems) {
  if (dirty_n_elems == N - 1) {
    // Actually means 0, so do the push.
    uint localBot = _bottom;
    elems[localBot] = t;
    OrderAccess::release_store(&_bottom, increment_index(localBot));
    return true;
  }
  return false;
}
```

Listing A.2: The `push_slow()` method of the ABP implementation.

```
pop_local(volatile E& t, uint threshold) {
  uint localBot = _bottom;
  // This value cannot be N-1.  That can only occur as a result of
  // the assignment to bottom in this method.  If it does, this method
  // resets the size to 0 before the next call (which is sequential,
  // since this is pop_local.)
  uint dirty_n_elems = dirty_size(localBot, _age.top());
  assert(dirty_n_elems != N - 1, "Shouldn't be possible...");
```

```
  if (dirty_n_elems <= threshold) return false;
  localBot = decrement_index(localBot);
  _bottom = localBot;
  // This is necessary to prevent any read below from being reordered
  // before the store just above.
  OrderAccess::fence();
  // g++ complains if the volatile result of the assignment is
  // unused, so we cast the volatile away.  We cannot cast directly
  // to void, because gcc treats that as not using the result of the
  // assignment.  However, casting to E& means that we trigger an
  // unused-value warning.  So, we cast the E& to void.
  (void) const_cast<E&>(t = _elems[localBot]);
  // This is a second read of "age"; the "size()" above is the first.
  // If there's still at least one element in the queue, based on the
  // "_bottom" and "age" we've read, then there can be no interference
  // with a "pop_global" operation, and we're done.
  idx_t tp = _age.top();    // XXX
  if (size(localBot, tp) > 0) {
    assert(dirty_size(localBot, tp) != N - 1, "sanity");
    TASKQUEUE_STATS_ONLY(stats.record_pop());
    return true;
  } else {
    // Otherwise, the queue contained exactly one element; we take the
    // slow path.
    return pop_local_slow(localBot, _age.get());
  }
}
```

Listing A.3: The `pop_local()` method of the ABP implementation.

```
pop_local_slow(uint localBot, Age oldAge) {
  // This queue was observed to contain exactly one element;
  // either this thread will claim it, or a competing "pop_global".
  // In either case, the queue will be logically empty afterwards.
  // Create a new Age value that represents the empty queue for
  // the given value of "_bottom". (We must also increment "tag"
  // because of the case where "bottom == 1", "top == 0".
  // A pop_global could read the queue element in that case, then
  // have the owner thread do a pop followed by another push. Without
  // the incrementing of "tag", the pop_global's CAS could succeed,
  // allowing it to believe it has claimed the stale element.)
  Age newAge((idx_t)localBot, oldAge.tag() + 1);
  // Perhaps a competing pop_global has already incremented "top", in
    which case it wins the element.
  if (localBot == oldAge.top()) {
    // No competing pop_global has yet incremented "top"; we'll try to
    // install new_age, thus claiming the element.
    Age tempAge = _age.cmpxchg(newAge, oldAge);
    if (tempAge == oldAge) {
      // We win.
      assert(dirty_size(localBot, _age.top()) != N - 1, "sanity");
      TASKQUEUE_STATS_ONLY(stats.record_pop_slow());
      return true;
    }
  }
  // We lose; a completing pop_global gets the element. But the queue
    is empty and top is greater than bottom. Fix this representation
    of the empty queue to become the canonical one.
  _age.set(newAge);
```

```
27    assert(dirty_size(localBot, _age.top()) != N - 1, "sanity");
28    return false;
29  }
```

Listing A.4: The `pop_local_slow()` method of the ABP implementation.

Appendix B

Spin Master of the ABP Implementation

```
1  bool do_spin_master_work(TerminatorTerminator* terminator) {
2    uint yield_count = 0;
3    // Number of hard spin loops done since last yield
4    uint hard_spin_count = 0;
5    // Number of iterations in the hard spin loop.
6    uint hard_spin_limit = WorkStealingHardSpins;
7
8    // If WorkStealingSpinToYieldRatio is 0, no hard spinning is done.
9    // If it is greater than 0, then start with a small number
10   // of spins and increase number with each turn at spinning until
11   // the count of hard spins exceeds WorkStealingSpinToYieldRatio.
12   // Then do a yield() call and start spinning afresh.
13   if (WorkStealingSpinToYieldRatio > 0) {
14     hard_spin_limit = WorkStealingHardSpins >>
       WorkStealingSpinToYieldRatio;
15     hard_spin_limit = MAX2(hard_spin_limit, 1U);
16   }
17   // Remember the initial spin limit.
18   uint hard_spin_start = hard_spin_limit;
19
20   // Loop waiting for all threads to offer termination or
21   // more work.
22   while (true) {
23     // Look for more work.
24     // Periodically sleep() instead of yield() to give threads
25     // waiting on the cores the chance to grab this code
26     if (yield_count <= WorkStealingYieldsBeforeSleep) {
27       // Do a yield or hardspin. For purposes of deciding whether
28       // to sleep, count this as a yield.
29       yield_count++;
30
31       // Periodically call yield() instead spinning
32       // After WorkStealingSpinToYieldRatio spins, do a yield() call
33       // and reset the counts and starting limit.
34       if (hard_spin_count > WorkStealingSpinToYieldRatio) {
35         yield();
36         hard_spin_count = 0;
37         hard_spin_limit = hard_spin_start;
38       } else {
39         // Hard spin this time
```

```
40          // Increase the hard spinning period but only up to a limit.
41          hard_spin_limit = MIN2(2*hard_spin_limit,
42                                 (uint) WorkStealingHardSpins);
43          for (uint j = 0; j < hard_spin_limit; j++) {
44            SpinPause();
45          }
46          hard_spin_count++;
47        }
48      } else {
49        yield_count = 0;
50
51        MonitorLockerEx locker(_blocker, Mutex::_no_safepoint_check_flag
    );
52        _spin_master = NULL;
53        locker.wait(Mutex::_no_safepoint_check_flag,
    WorkStealingSleepMillis);
54        if (_spin_master == NULL) {
55          _spin_master = Thread::current();
56        } else {
57          return false;
58        }
59      }
60
61      size_t tasks = tasks_in_queue_set();
62      bool exit = exit_termination(tasks, terminator);
63      {
64        MonitorLockerEx locker(_blocker, Mutex::_no_safepoint_check_flag
    );
65        // Termination condition reached
66        if (_offered_termination == _n_threads) {
67          _spin_master = NULL;
68          return true;
69        } else if (exit) {
70          if (tasks >= _offered_termination - 1) {
71            locker.notify_all();
72          } else {
73            for (; tasks > 1; tasks--) {
74              locker.notify();
75            }
76          }
77          _spin_master = NULL;
78          return false;
79        }
80      }
81    }
82  }
```

Listing B.1: Termination algorithm for the spin master thread [25].

The spin master method in listing B.1 shows what steps the chosen spin master thread takes to coordinate the other threads' termination efforts. It basically checks whether all threads are either done or if any deque still has tasks in it. In the latter case, the spin master wakes up other threads, depending on the amount of available tasks, so they can steal those available tasks.

In lines 13 to 15, the limit for how often the spin master will hard spin, before it yields, gets calculated. Then in line 22, the loop to check the queues for remaining tasks starts. The lines 34 to 46 decide if either a yield, or a hard spin should be done. The number of hard spins increases each time until it hits a certain limit.

Then a yield happens and the hard spin counter is reset. After several yields, the current spin master gives other threads the chance to become the new spin master in lines 49 to 57. In line 61, the spin master determines the number of tasks that are still in other thread's queues. Then, in lines 66 to 68, the spin master might leave if all other threads are done. Otherwise, in lines 69 to 78, it wakes up threads that are waiting in the `offer_termination()` method so they can take over unfinished tasks. However, it will not wake up more threads than tasks are available. Finally, the spin master resigns.

Appendix C

Median Pause Times of all Benchmark Runs

		Median pause times [ms]		
			Idempotent work stealing	
	Benchmark	ABP	Double-ended	LIFO
DaCapo	h2	298.935	291.216	285.975
	jython	2.294	2.226	2.107
	pmd	11.064	12.222	13.565
	tomcat	7.118	7.340	6.567
	tradebeans	229.587	207.622	199.158
	xalan	3.441	2.868	2.953
DaCapo Scala	apparat	24.948	23.639	23.151
	factorie	140.297	125.050	126.268
	scalac	34.248	31.012	28.370
	scalatest	12.472	10.891	12.529
	scalaxb	9.371	8.125	7.377
	tmt	8.828	8.415	9.397
SPECjvm	scimark.lu.large	50.532	50.327	47.670
	xml.validation	39.693	40.656	47.515

Table C.1: Median pause times of the APB and idempotent work stealing benchmark runs.

		Median pause times [ms]			
		Task-pushing old task dripping			
	Benchmark	N1	N2	N4	N8
DaCapo	h2	278.089	285.937	282.041	284.443
	jython	2.349	2.219	2.372	2.327
	pmd	16.905	14.154	16.942	16.096
	tomcat	7.933	7.715	7.525	7.360
	tradebeans	197.277	200.182	202.117	201.231
	xalan	3.363	3.253	3.212	3.047
DaCapo Scala	apparat	23.194	24.228	33.994	33.054
	factorie	116.584	137.560	126.356	123.761
	scalac	30.474	30.518	31.916	32.827
	scalatest	11.682	13.641	12.526	14.144
	scalaxb	8.678	10.037	17.982	12.792
	tmt	8.966	10.094	10.411	10.414
SPECjvm	scimark.lu.large	50.785	52.403	50.182	52.851
	xml.validation	36.095	37.690	39.215	39.124

Table C.2: Median pause times of the task-pushing benchmark runs.

Sworn Declaration

I hereby declare under oath that the submitted Master's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Vienna, May 2021