Garbage collection

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
Algorithms
A **heap**: of one or more blocks of contiguous words

A **object**: a heap-allocated contiguous region addressed by 0+ pointers

A **mutator**: application thread, opaque to the collector except for heap operations (allocate, read, write)

A **root**: a heap pointer accessible to the mutator
(e.g. in static global storage, stack space, or registers)

An object is **live** if a mutator will access it in the future

An object is **reachable** if there is a chain of pointers to it from a root
Mark

\[
\text{mark}(\text{node}) = \\
\quad \text{if not node.marked:} \\
\quad \quad \text{node.marked} = \text{True} \\
\quad \text{for } c \text{ in node.children:} \\
\quad \quad \text{mark}(c)
\]
Mark

```python
mark(node) =
    if not node.marked:
        node.marked = True
        for c in node.children:
            mark(c)
```
Mark

mark(node) =
  if not node.marked:
    node.marked = True
    for c in node.children:
      mark(c)
Mark-and-sweep collection

Mark

mark(node) =
    if not node.marked:
        node.marked = True
        for c in node.children:
            mark(c)

root
Mark

mark(node) =
    if not node.marked:
        node.marked = True
        for c in node.children:
            mark(c)
Mark-and-sweep collection

Mark

mark(node) =
    if not node.marked:
        node.marked = True
        for c in node.children:
            mark(c)
Mark-and-sweep collection

Algorithms

Mark

```
mark(node) =
    if not node.marked:
        node.marked = True
        for c in node.children:
            mark(c)
```
Mark-and-sweep collection

Mark

mark(node) =
    if not node.marked:
        node.marked = True
    for c in node.children:
        mark(c)
Mark

mark(node) =
    if not node.marked:
        node.marked = True
        for c in node.children:
            mark(c)
Collect

- **copy** live blocks to to-space (starting at the root)
- leave **forwarding addresses** in from-space
- **switch roles** of spaces

**from-space**

**to-space**
Collect

- **copy** live blocks to to-space (starting at the root)
- leave **forwarding addresses** in from-space
- **switch roles** of spaces
Collect

**copy** live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
**switch roles** of spaces
Collect

copy live blocks to to-space (starting at the root)
leave forwarding addresses in from-space
switch roles of spaces
Collect

**copy** live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
**switch roles** of spaces
Collect

**copy** live blocks to to-space (starting at the root)
leave **forwarding addresses** in from-space
**switch roles** of spaces
Collect

- **copy** live blocks to to-space (starting at the root)
- leave **forwarding addresses** in from-space
- **switch roles** of spaces
The **reference count** tracks the number of pointers to each object.

An object’s reference count is 1 when the object is created:

The count is incremented when a pointer newly references the object:

The count is decremented when a pointer no longer references the object:

The object is unreachable garbage when the reference count goes to 0:
Motivation: collector has imperfect information about object layout (e.g. because language is compiled to C)

Idea: use an approximation to guess whether a value represents a pointer, e.g.:

1. does the value point into the heap?
2. does it point to valid metadata?

Drawbacks

1. (chance) can incorrectly classify addresses as pointers
2. (subterfuge) can fail to identify disguised pointers
Generational collection

Copying collector for minor heap / mark-and-sweep for major heap
Generational collection

Copying collector for minor heap / mark-and-sweep for major heap

- minor heap
- major heap
- root
Copying collector for minor heap / mark-and-sweep for major heap
Copying collector for minor heap / mark-and-sweep for major heap
Copying collector for minor heap / mark-and-sweep for major heap
Generational collection

Copying collector for minor heap / mark-and-sweep for major heap

minor heap

root

major heap
Copy of collector for minor heap / mark-and-sweep for major heap

root

minor heap

major heap
Copying collector for minor heap / mark-and-sweep for major heap
Copying collector for minor heap / mark-and-sweep for major heap
Copying collector for minor heap / mark-and-sweep for major heap
Generational collection

Copying collector for minor heap / mark-and-sweep for major heap

- minor heap
- major heap
- root
Generational collection

Copying collector for minor heap / mark-and-sweep for major heap
Generational collection

Algorithms

Performance

Reading

Copying collector for minor heap / mark-and-sweep for major heap

minor heap

major heap

root
Generational collection

Copying collector for minor heap / mark-and-sweep for major heap
Performance
**Throughput**: mutator performance

**Latency**: pauses in mutator execution

**Space overhead**: e.g. due to mark bits, layout information

**More** (combination of program behaviour and collector design):

- maximum heap size
- allocation rate
- collection frequency
- mean object size
- proportion of heap occupied by large objects
Example
Pause times alone provide little information.
A good distribution of pause times is needed for mutators to make progress.

Example
Compaction can slow collection but improve locality (& hence throughput)
Many mature systems combine several standard algorithms.

For example, Cedar (1985):

“[…] provides both a **concurrent reference-counting collector** that runs in the background when needed, and a **pre-emptive conventional “trace-and-sweep” collector** that can be invoked explicitly by the user to reclaim circular data structures […]

“Both collectors treat procedure-call activation records (called frames) **“conservatively”**; that is they assume that every ref-sized bit pattern found in a frame might be a ref”
Reading
A Real-time Garbage Collector with Low Overhead and Consistent Utilization

David F. Bacon
dfb@watson.ibm.com
IBM T.J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598

Perry Cheng
perryche@us.ibm.com
IBM T.J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598

V.T. Rajan
vtrajan@us.ibm.com
IBM T.J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598

ABSTRACT

Note that the use of garbage collection in languages like Java is becoming more prevalent and critical as systems become more complex. Garbage collection is a necessary evil in real-time systems. For approaches that have previously allowed live copying to ensure no data loss, short individual pause times, but are unable to achieve consistent mutator CPU utilization.

The paper is organized as follows: Section 2 describes previous work. Section 3 describes an overview of our design. Section 4 presents an overview of our algorithm. Section 5 discusses our implementation details. Section 6 presents experimental results. Section 7 presents conclusions.

Keywords

Real-time garbage collection, real-time scheduling, utilization

1. INTRODUCTION

Garbage-collected languages like Java are making significant inroads and dominate in BEDE data. However, the engineering and problems that arise from the simplicity of concurrency models is that they are subject to a number of inherent limitations. A system that is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with garbage collection, programming with garbage collection remains an unsolved issue in the worst-case behavior of real-time systems, where short individual pause times, but are unable to achieve consistent mutator CPU utilization. Our collector is unique in that it occupies an under-explored portion of the design space for real-time garbage collection.

“[… there is significant interest in applying garbage collection to hard real-time systems.”

“Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds.”

“We […] show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application.”

Concurrent GCs and Modern Java Workloads: A Cache Perspective

Maria Carpen-Amarie
Huawei Zurich Research Center
maria.carpen.amarie@huawei.com

Georgios Vavouliotis
Huawei Zurich Research Center
gvavouliotis2@huawei.com

Konstantinos Tovletoglou
Huawei Zurich Research Center
k.tovletoglou@huawei.com

Boris Grot
University of Edinburgh
b.grot@ed.ac.uk

Rene Mueller
Huawei Zurich Research Center
renemueller@huawei.com

Abstract

The garbage collector (GC) is a crucial component of language runtimes, offering convenience, generational and high productivity in exchange for a run-time overhead. Concurrent collection mechanism applications through parallelism and shared CPU resources. A likely point of contention between mutators and GC behavior is, consequently, a significant source of productivity loss.

Even for these, the performance overhead of GC does not exceed 10%. This work builds on the hypothesis that the cache pollution caused by concurrent GCs hurts application performance. We validate this hypothesis with a cache sensitive micro-benchmark. We find that concurrent GC activity may slow down the application by up to 3× and increase the LLC access by a factor of almost 7×. However, the benefits offered by the GC do not come for free. 4.5% of benchmarks show a statistically significant correlation between GC-induced cache pollution and performance. In the present work, we study the performance overhead source is the shared last-level cache (LLC).

1 Introduction

Automatic memory management, also known as garbage collection (GC), is a technique that provides memory access safety and reliability while significantly reducing development load. These aspects inside garbage collection are essential components of many software environments (e.g., Java, C#, JavaScript), which are intensively used by web services (e.g., Twitter), web browsers, and mobile platforms (e.g., Android). For these reasons, automatic memory management continues to be a hot topic today, even after more than half a century of active research towards its optimization.

However, the benefits offered by the GC do not come for free. Prior work [1–7, 13, 19, 24, 30] has shown that application performance is profoundly contingent on the effectiveness of the garbage collector. Historically, GCs harmed application performance due to uncontrollably long stop-the-world (STW) pauses to reduce the performance overhead. The current generation of GCs have largely moved away from this model, advocating incremental collection (GC), 3×, ZGC [12–16], G1 [17] and, therefore, most modern GCs [18]. Even with increasingly shorter pauses, while performance meets the needs of commodity applications with the appearance (e.g., ZGC [12]) makes the assumption of a processor as an ILP, the application effort in shared resources with the GC (e.g., cache capacity, bandwidth, CPU limit and even sometimes help with the collection itself). To the best of our knowledge, there is no work on concurrent GCs that quantifies these overhead components individually. Such an effort would be valuable for both researchers and facilitate well-targeted performance improvements.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM ISBN 979-8-4007-0179-5/23/06... $15.00

ACM Reference Format:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

This work builds on the hypothesis that the cache pollution caused by concurrent GCs hurts application performance.

“We find that concurrent GC activity may slow down the application by up to 3× and increase the LLC misses by 3 orders of magnitude.”

“However, [...] we find that only 5 out of 23 benchmarks show a statistically significant correlation between GC-induced cache pollution and performance.”
Introduction

Modern concurrent garbage collectors are surprisingly expensive to implement and maintain. Applications and systems that depend on low pause times and high throughput often choose collectors such as C4, Shenandoah, and ZGC built on the G1 foundation, seeking to further reduce memory footprints and latency-sensitive applications led vendors to focus on low pause time collectors such as C4 [3], Shenandoah [2], and ZGC [1]. While these approaches deliver low pause times, they do so at high memory and CPU cycles. Furthermore, no evidence that low pause times do not always translate into low latency for latency-sensitive applications. This paper identifies the key issues expensive and proposes a very different approach. We introduce LXR (Low-Latency, High-Throughput Garbage Collection) and propose it in OpenJDK, and compare it against these widely-used collectors on diverse contemporary workloads.

The early Garbage First (G1) collector is a copying collector. G1 reclaims space, delivering high throughput and excellent pausing times. G1 is a hierarchical Immix heap, with 10 levels of maturity, and a large garbage collection heap. Immix heap is a baseline heap for G1. G1 uses a coarsely-grained, low-overhead write barrier. These choices have fundamental implications on latency-critical Immix workloads. LXR combines excellent responsiveness and throughput, while imposing a low processing overhead. LXR identifies cyclic garbage. LXR introduces: i) RC (Reference Counting) and judicious copying. RC delivers scalability and maintainability, primarily by avoiding young and mature objects. RC is a hierarchical Immix heap structure, with a lowest level as a fast reference counting heap; ii) evacuation with concurrent tracing, an intrinsically expensive approach, but with concurrent tracing, an intrinsically expensive approach, but with concurrent tracing, an intrinsically expensive approach, but with concurrent tracing, an intrinsically expensive approach. RC, in a hierarchical Immix heap structure, reclaims most memory without any copying. Occasional concurrent collectors such as C4, Shenandoah, and ZGC use substantially more CPU cycles and memory than simpler collectors.

This paper [...] uses the insight that regular, brief stop-the-world collections deliver sufficient responsiveness at greater efficiency than concurrent evacuation.

“ [...] LXR delivers 7.8× better throughput and 10× better 99.99% tail latency than Shenandoah.”

Paper 3: Zhao et al (2022)