Garbage Collection Optimization for Non Uniform Memory Access Architectures

Khaled Abdulrahman Alnowaiser

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

June 2016

Abstract

Cache-coherent non uniform memory access (ccNUMA) architecture is a standard design pattern for contemporary multicore processors, and future generations of architectures are likely to be NUMA. NUMA architectures create new challenges for managed runtime systems. Memory-intensive applications use the system's distributed memory banks to allocate data, and the automatic memory manager collects garbage left in these memory banks. The garbage collector may need to access remote memory banks, which entails access latency overhead and potential bandwidth saturation for the interconnection between memory banks.

This dissertation makes five significant contributions to garbage collection on NUMA systems, with a case study implementation using the Hotspot Java Virtual Machine.

It empirically studies data locality for a Stop-The-World garbage collector when tracing connected objects in NUMA heaps. First, it identifies a locality richness which exists naturally in connected objects that contain a root object and its reachable set— 'rooted sub-graphs'. Second, this dissertation leverages the locality characteristic of rooted sub-graphs to develop a new NUMA-aware garbage collection mechanism. A garbage collector thread processes a local root and its reachable set, which is likely to have a large number of objects in the same NUMA node. Third, a garbage collector thread *steals* references from *sibling* threads that run on the same NUMA node to improve data locality.

This research evaluates the new NUMA-aware garbage collector using seven benchmarks of an established real-world DaCapo benchmark suite. In addition, evaluation involves a widely used SPECjbb benchmark and Neo4J graph database Java benchmark, as well as an artificial benchmark. The results of the NUMA-aware garbage collector on a multi-hop NUMA architecture show an average of 15% performance improvement. Furthermore, this performance gain is shown to be as a result of an improved NUMA memory access in a ccNUMA system.

Fourth, the existing Hotspot JVM adaptive policy for configuring the number of garbage collection threads is shown to be suboptimal for current NUMA machines. The policy uses outdated assumptions and it generates a constant thread count. In fact, the Hotspot JVM still uses this policy in the production version. This research shows that the optimal number of garbage collection threads is application-specific and configuring the optimal number of garbage collection threads yields better collection throughput than the default policy. Fifth, this dissertation designs and implements a runtime technique, which involves heuristics from dynamic collection behavior to calculate an optimal number of garbage collector threads for each collection cycle. The results show an average of 21% improvements to the garbage collection performance for DaCapo benchmarks.

Acknowledgements

All my praises be to Allah, the almighty, for granting me the strength and patience to complete this research successfully.

I would like to express my deepest gratitude and appreciation to my supervisor, Dr. Jeremy Singer, for his encouragements, guidance, and support. I am grateful for his advice and inspiration. I also extend my gratitude to Dr. Paul Cockshott for all the interesting and challenging discussion, which helped me shaping this research in a coherent way.

I am grateful to my sponsor, Prince Sattam bin Abdulaziz University for funding this PhD research. I would like to thank the support team and colleagues in the School of Computing Science for providing me with all help to complete this research.

Finally and most importantly, I offer my gratefulness to my family for their constant care, encouragement, and support throughout my life. I am thankful to my wife and children for their support and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

AMD Advanced Micro Devices

CPU Central Processing Unit

DCT DRAM Controller

DSM Distributed Shared Memory

JNI Java Native Interface

JVM Java Virtual Machine

LLC Last Level Cache

MCT Memory Controller

NUMA Non-Uniform Memory Access

PLAB Promotion Local Allocation Buffer

QPI QuickPath Interconnect

SMP Symmetric Multiprocessor Chip

SRI System Request Interface

THP Transparent Huge Page

TLB Translation Lookaside Buffer

TLAB Thread Local Allocation Buffer

UMA Uniform Memory Access

CHAPTER

1

INTRODUCTION

This chapter introduces the context of garbage collection optimization for non-uniform memory access architectures. It also presents the motivation to carry out this research followed by the thesis statement. The chapter reports the main contributions and publications of this research and concludes by outlining the dissertation structure.

1.1 Overview

The revolution in semiconductor technologies, marked by the prevalence of *multicore* processors, has created high computational capacity resources to enhance program performance. Programmers split the code into several segments and use a parallel programming model to enable the execution of some segments in parallel. Today, data-intensive applications, for example databases, data analytic engines, and application servers, have consumed available CPU and memory resources on server-class machines. This increasing growth in applications' computation needs has demanded a rapid expansion on hardware resources.

Since the transition from unicore to multicore processors [Sutter and Larus, 2005], the core counts have increased consistently and an optimistic forecast suggests that the number of cores will continue to follow Moore's law [Moore, 2000] (*the number of transistors is doubled every 18 months*). However, memory has an unequal technological development pace as compared to multicore processors. The processor-memory performance gap means that

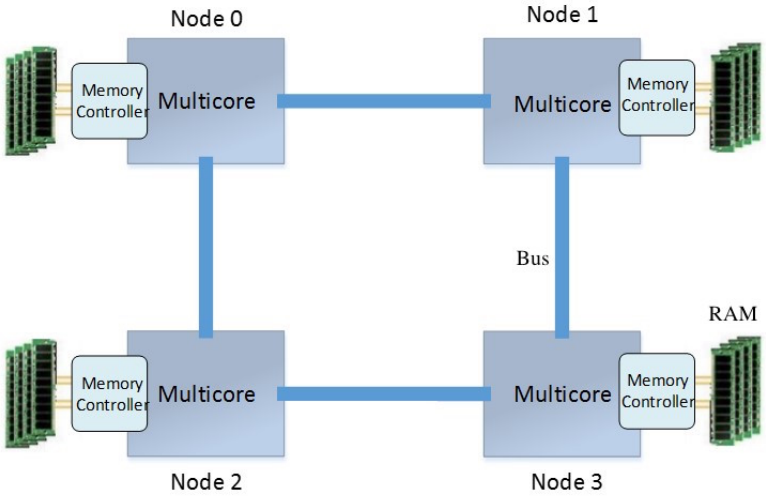


Figure 1.1: An example of a NUMA architecture

merely adding more cores to the processor does not improve program performance automatically because the memory bandwidth is unable to handle the increased memory traffic. As a result, this multicore architecture has hit the "memory wall" [McKee, 2004] and limited the core counts. Therefore, increasing the memory bandwidth is a big challenge for multicore processor designers.

One way to augment the memory bandwidth and reduce access latency is to distribute the memory physically across the processors while maintaining a shared address space. This architecture is called Distributed Shared Memory (DSM). Processors can access any memory address, however, the access latency depends on the distance between data in memory and the processor accessing it. Although DSM systems usually comprise a network of machines, DSM can also be implemented in a single machine. This architecture is referred as Non-Uniform Memory Access (NUMA) architecture [Hennessy and Patterson, 2011b].

NUMA systems involve multiple Central Processing Unit (CPU) *sockets*, each has a multicore processor chip. In this design, sockets are connected through a network of high speed links, e.g. Intel QuickPath Interconnect (QPI) [Intel, 2016] and AMD HyperTransport technologies [AMD, 2016b]. A multicore processor and its memory forms a **NUMA node**, where cores of the same NUMA node incur symmetric memory access latency to the local memory and asymmetric access latency to the "remote" memory. Figure 1.1 depicts an example of multicore-based multiprocessor NUMA architecture.

The implications of NUMA architecture for software design/development are significant. A multi-threaded application may exhibit performance degradation when running on multiple NUMA nodes. Application data can be allocated in any NUMA node and threads may need to access remote memory to execute the code. A non NUMA-aware memory allocation policy could impair application performance by raising off-node communications and

increasing access latency, saturating some interconnection links, or by putting pressure on the local-node's memory hierarchy due to imbalanced allocation. There are several tools to monitor these events (Chapter 4 describes the LIKWID tool).

Cache coherency for prevalent NUMA architectures participates into two issues. First, cache coherency limits scaling up the number of cores due to the difficulties of managing cached data across large number of cores. Second, a ccNUMA system incurs performance challenge when multiple cores share a cache line but not data in the cache line. This "false sharing" necessitate cache invalidation and cause performance degradation. However, this dissertation shows that NUMA remote accesses are more serious problem than cache coherency. Section 6.4.3 discusses this issue in more detail.

Managed runtime systems, for example the Java Virtual Machine (JVM), abstract low-level details such as memory management and hardware configuration. However, many runtime system deployments manage program execution in a NUMA-agnostics fashion. In fact, they usually devolve memory management, thread scheduling, and/or other components to the operating system. Operating system's tools for NUMA management, for example memory allocation policies, could be inefficient without programmer's intervention. The default memory allocation policy is to allocate memory from the NUMA node of first core touching it, however, applications could involve imbalanced memory allocations between nodes, which may cause some node to saturate. Therefore, higher level of NUMA management would be required to explicitly manage program execution.

Garbage collection is a performance critical component of managed runtime systems. A garbage collector reclaims the memory occupied by dead (unreachable) objects, which is no longer needed by the application. Detecting unreachable objects can be done by identifying the reachable objects. To identify and preserve the reachable objects, a garbage collector traverses the reference graph starting from "root" objects, for example global and static variables. The reference graph may contain references to objects that are distributed across the NUMA nodes. Therefore, the garbage collection threads may incur additional overhead when accessing remote NUMA nodes. Furthermore, a copying and a compacting garbage collector could relocate a reachable object to a different NUMA node. Data locality, which is keeping data close to the core accessing it, could be changed due to the relocation operation by the garbage collector. Consequently, application threads would access remote data, causing the performance to degrade.

The goal of this research is to investigate improvements to the garbage collection of the Hotspot JVM running on NUMA architecture. Previous research has reported inefficient garbage collection performance when running on NUMA machines, for example [Gidra et al., 2011]. There exist several techniques to improve NUMA garbage collection performance [Tikir and Hollingsworth, 2005, Ogasawara, 2009, Gidra et al., 2015]. However,

these techniques use a common mechanism, which involves inspecting every reachable object's location before processing it. This mechanism may require a complex heap layout or expensive memory access samples. In addition, these techniques utilize the system's full computation resources to execute the garbage collection workload. In the NUMA context, consuming all cores for garbage collection may increase off-node traffic, hence degrading the garbage collection performance.

This dissertation provides novel approaches to improve NUMA garbage collection. Instead of inspecting every reachable object, this research proposes inspecting a small set of the reachable objects, *the root set*. This research hypothesizes that the majority of references in the transitive closure of a root reference reside in the same NUMA node as the root (Section 5.2).

By exploiting this locality characteristic, garbage collection performance gains improvement. In addition, this research proposes a runtime garbage collection thread management policy. This policy responds to the changes of collection performance by dynamically changing the number of garbage collection threads. These *NUMA-aware* techniques are shown to improve the garbage collection performance.

1.2 Thesis Statement

Given that NUMA systems partition the memory into multiple nodes, and a multi-threaded application can allocate data in any NUMA node, parallel garbage collection involves off-node communications costs when collecting garbage memory. This research asserts that NUMA topology awareness can improve garbage collection performance. By obtaining data location, garbage collection threads can process NUMA-local data. In addition, NUMA congestion caused by the increased number of scheduled garbage collection threads can be alleviated by dynamically adapting the number of threads.

This assertion is demonstrated by the following:

- The implementation of a NUMA-aware garbage collector, which takes into account the object location when copying/promoting objects in a generational NUMA heap.
- The implementation of adaptive garbage collection thread management policy that dynamically adapts the number of scheduled threads based on the collection throughput.
- An evaluation of the NUMA-aware garbage collector and the adaptive garbage collection thread management policy on seven benchmarks of an established real-world DaCapo benchmark suite, a widely used SPECjbb benchmark, Neo4J graph database Java benchmark, and an artificial benchmark.

1.3 Contributions

This work contributes to NUMA-based garbage collection in a number of ways:

- Development of the rooted sub-graph hypothesis [Alnowaiser, 2014], which states that a high proportion of objects in a root's transitive closure reside in the same NUMA node as the root (Chapter 5). This hypothesis is evaluated and the results show that 80% of objects reside in the same node as root. The introduction of rooted sub-graph notion and its use to make garbage collection NUMA-aware is a major contribution of this research.
- Development of root reference classification and distribution strategy, which enables garbage collection threads to process NUMA-local rooted sub-graphs [Alnowaiser and Singer, 2016] (Chapter 6).
- Development of a NUMA-local work stealing strategy to allow garbage collection threads to steal from NUMA local queues [Alnowaiser and Singer, 2016] (Chapter 6).

  Our new garbage collection that uses NUMA-aware root classification and NUMA-local work stealing performs 15% on average better than the default Hotspot JVM.
- An investigation of the impact of scheduling large number of garbage collection threads on collection throughput and NUMA off-node traffic (Chapter 7).
- Development of an adaptive garbage collection thread policy to schedule an appropriate number of threads (Chapter 7).

  Our adaptive garbage collection thread management policy shows 21% and 5% on average performance improvement for DaCapo and overall benchmarks, respectively.

1.4 Publications

The work presented in this dissertation has led to the following publications:

1. Khaled Alnowaiser. A Study of Connected Object Locality in NUMA Heaps. In Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC 14, pages 1:1–1:9, New York, NY, USA, 2014. ACM. `http://dx.doi.org/10.1145/2618128.2618132`. [Alnowaiser, 2014]
2. Khaled Alnowaiser and Jeremy Singer. Topology-Aware Parallelism for NUMA Copying Collectors, chapter Languages and Compilers for Parallel Computing: 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, pages

191–205. Springer International Publishing, http://dx.doi.org/10.1007/978-3-319-29778-1_12. [Alnowaiser and Singer, 2016]

We intend to publish the outcomes of Chapter 7, which is the adaptive garbage collection thread management policy, in the near future.

1.5 Thesis Outline

The remainder of this dissertation is organized as follows:

Chapter 2: Literature Survey

This chapter reviews related work that focuses on optimizing garbage collection. Various approaches to deal with garbage collection in NUMA architectures, from graph traversal order, object locality, heap partitioning, to several data placement policies are discussed. My approach in this chapter is to combine description, discussion, and scrutiny of related work on the context of this dissertation. This approach would shed light on my motivation to carry out this research.

Chapter 3: Technical Background

This chapter provides a technical background on hardware and software systems used in this research. It refers to manufacturers' manuals, white papers, and illustrations explaining processors and memory management in the Linux operating system. It also presents and discusses NUMA architectural design and implementation for AMD processors. In addition, this chapter includes detailed description of the garbage collection policies (including the copying collector, *minor collection* and the mark-compact collector *major collection*) in the Hotspot JVM of OpenJDK.

Chapter 4: Experimental System Infrastructure

The experimental system infrastructure used for the work in this dissertation is fixed to one hardware and one software platform. In this chapter, I describe these system configurations. The work in this dissertation has been evaluated with seven benchmarks of an established real-world DaCapo benchmark suite, a widely accepted SPECjbb benchmark, Neo4J graph database Java benchmark, and an artificial benchmark. For every benchmark, this chapter describes the program and its memory allocation behavior. In addition, it show some of the configurable options that are used in the experimentation.

Chapter 5: A Study of Reference Locality

Chapter five reports an observational study for connected object's locality when garbage collector threads traverse the reference graph. Based on the results of this study, this chapter (Chapter 6) develops the rooted sub-graph hypothesis, which is the basis for my optimizations presented in this research.

Chapter 6: NUMA-Aware Garbage Collector

This chapter utilizes rooted sub-graph hypothesis to improve garbage collection performance. It applies NUMA awareness to the copying collector with three optimization schemes. These schemes aim to enhance NUMA locality by processing NUMA-local objects and reduce NUMA congestion by utilizing off-node resources. Heap scalability is also discussed with reference to the proposed optimization schemes.

Chapter 7: NUMA-Aware Garbage Collection Thread Management

This chapter investigates the correlations between the number of scheduled garbage collection threads and collection throughput. It also investigates NUMA congestion when varying the number of garbage collection threads. The results lead to the creation of NUMA-aware garbage collection thread management policy. This policy tracks collection throughput and adapts, if needed, the number of threads at runtime.

Chapter 8: Conclusion

Chapter eight concludes the work presented in this thesis and explores opportunities for further work.

Part I

STATE OF THE ART

CHAPTER

2

LITERATURE SURVEY

The garbage collector accesses memory intensively to reclaim dead memory and preserve live objects that the application needs. This excessive access incurs latency overhead due to the fact that the garbage collection exhibits poor temporal and spatial locality behavior [Jones et al., 2011]. In fact, the garbage collector is shown to get worse locality behavior when the heap is created in a system with NUMA architecture. The JVM can allocate objects in any NUMA node. As the program execution pauses to reclaim memory, the parallel garbage collection treats the live object set as a graph and each collector thread processes multiple sub-graphs. Objects in a sub-graph could be dispersed across different NUMA nodes; causing the garbage collector to pay additional remote memory access overhead. In the context of generational heaps, objects may be relocated to new addresses, possibly on remote NUMA nodes. When the program resumes execution, the new object layout could impose remote accesses; hence causing additional latency overhead.

This chapter reviews state of the art garbage collection optimization on NUMA architectures. It presents locality improvement research for garbage collection in a hierarchical order from a cache line to the virtual memory space. This presentation includes the discussion of the effect of improving object locality, in the virtual memory space, on the physical memory space represented by NUMA architecture. Furthermore, this chapter discusses the advantages and disadvantages of existing NUMA heap optimizations and identifies gabs that form the basis of this research.

The outline of this chapter is as follows. Section 2.1 introduces this chapter with basic

garbage collection algorithms. Section 2.2 presents parallel techniques applied to garbage collection. Section 2.5 surveys various optimization schemes to improve object locality. Object segregation techniques are reviewed in Section 2.6. Section 2.3 investigates several heap partitioning strategies, whereas Section 2.4 explores *NUMA* partitioning schemes and related techniques to improve object locality. Section 2.7 discusses various data placement policies and Section 2.8 summarizes this chapter.

2.1 Basic Garbage Collection Algorithms

The virtual machines reserve a space for the heap which contains program's dynamically allocated data. A program may refer to any process that requires heap space. This includes the kernel, file managers, network daemons, runtime systems and user applications. In this dissertation, programs refer to Java-based applications. During program execution, threads *mutate* the heap by allocating new objects or changing the connectivity between objects. When a running program abandons access to some objects in the heap, those objects are considered *garbage* and they become subject to reclamation. The space occupied by those "dead" objects should return to the program for reuse or to the operating system.

The basic garbage collection functions consist of two parts [Wilson, 1992]:

1. It must distinguish between the live objects and the dead objects.
2. It reclaims the dead memory and makes it available to the program to use it.

Liveness is a global criterion that garbage collection uses to identify live objects. When program execution pauses for garbage collection, the set of values that a program can manipulate directly are those held in the processor registers, those on the program stack that are in the stack frames (these include the local variables), and those held in the global variables. These globally visible variables are called the *root set*. Heap objects that are directly reachable from the root set or indirectly by traversing pointers from the root set are considered live and must be preserved. Therefore, the *live object set* is treated as a directed path graph, where nodes denote live objects and edges denote references. Any other unreachable object is considered garbage and its space can be safely reclaimed. This is a conservative estimate of live objects.

There are four algorithms, in which any garbage collection scheme relies on: mark-sweep collection, mark-compact collection, copying collection, and reference counting [Jones et al., 2011]. The following sections present the basic algorithms of these four garbage collection schemes. They are meant to give a brief description of their sequential implementation prior to consider parallel versions when running on multicore processors.

2.1.1 Mark-Sweep Collection

Mark-sweep collection [McCarthy, 1960] operates recursively on the reference graph to *mark* reachable objects as live and *sweep* unmarked objects. The two basic functions of mark-sweep algorithm are as follows:

- **Garbage Detection**: Starting from the root set, garbage collection traverses the reference graph and objects that are reachable from the root set are marked live. Marking is done by altering a field in the object header or using a bitmap side table.
- **Garbage Reclamation**: Unmarked objects are swept and their space is recycled and returned to the allocator for reuse.

Memory fragmentation is a major problem with mark-sweep collection. Recycled space of small object size may not be continuous to fit larger objects. In addition, object temporal locality may change because new objects will be allocated in the reclaimed memory adjacent to different-age objects. Section 2.5.1 describes previous research on improving object temporal locality. In the context of NUMA architecture, new objects may scatter across various NUMA nodes. In addition to the poor temporal locality, garbage collection threads may incur additional overhead to process objects in remote NUMA nodes.

2.1.2 Mark-Compact Collection

Mark-compact collection [Saunders, 1964] overcomes the problem of fragmentation occurred to mark-sweep collector. Live objects are moved and *compacted* into a continuous space. The remaining "continuous" free space is returned to the allocator. Garbage detection and collection is as follows:

- **Garbage Detection**: The marking phase traces the reference graph and marks the reachable objects.
- **Garbage Reclamation**: Live objects are relocated and compacted such that they become adjacent to the other live objects.

Object locality of the compaction order is important. Arbitrary order compaction does not consider the original order or object connectivity, which may lead to poor spatial locality. Alternatively, sliding compaction keeps object order as allocated by mutator threads. Modern mark-compact collectors implement sliding compaction [Jones et al., 2011].

The major disadvantage of compaction collection is the need for multiple passes over the live object set including the marking pass and the sliding pass. These multiple passes increase the

time overhead. In addition, compaction for NUMA heaps is significant because object spatial locality is likely to change after sliding live objects to heap sides. Section 2.4 describes various improvements to object spatial locality.

2.1.3 Copying Collection

Copying collection remedies the heap fragmentation by moving the live object set to a contiguous area. In contrast to the compaction collection, copying collection requires only one pass over the live objects. In a semispace copying collector [Cheney, 1970b], live objects in one space are moved to the other space, making subsequent allocations fast. The main disadvantage is, it reduces the heap size by half and may change object locality. However, several studies take advantage of moving objects by improving object temporal locality, for example Huang et al. [2004], and spatial locality, for example Gidra et al. [2013] .

- **Garbage Detection**: The copying collector traverses the reference graph and it moves an object to its new location as the collector reaches it (one pass).
- **Garbage Reclamation**: Once the live objects *scavenged* to the new space, the old space is recycled and reused by the allocator.

2.1.4 Reference Counting

Reference counting algorithm operates directly on the heap objects to identify liveness property of each object [Collins, 1960]. Instead of traversing the reference graph to determine the live object set and infer garbage objects, each object has a counter that is incremented/decremented whenever a reference to that object is created or destroyed. The basic functions of reference counting algorithm are as follows:

- **Garbage Detection**: Since each object has a reference counter to keep track of the amount of references to it, any object contains one or more references in its reference counter is considered as live.
- **Garbage Reclamation**: Objects with zero reference are no longer reachable by the running program and are subject to reclamation. Furthermore, when reclaiming a dead object, garbage collection must decrement reference counters of objects referenced by the dead object. This process may propagate through the reachable set of the reclaimed object.

Reference counting distributes memory management costs throughout program execution. In addition, it can reclaim memory as soon as an object becomes dead. However, reference counting algorithm has several disadvantages. Firstly, it is unable to collect cyclic data structures, which contain references to themselves, for example doubly-linked lists [McBeth, 1963]. Self-referential data structures are common in programming languages, although their frequency varies between applications [Bacon and Rajan, 2001]. Second, reference count manipulations must be atomic in order to avoid race conditions between mutator threads. Third, the mutator threads exhibit time overhead when manipulating reference counters. Fourth, the references size that a reference counter may have could be equal to the number of objects in the heap. This storage overhead can be significant in today's application large heap size.

There are several approaches to solve some of reference counting problems, for example combining tracing algorithms with reference counting [Blackburn and McKinley, 2003]. The next section will review the parallel garbage collection algorithms, which are implemented to collect multicore processors.

2.2 Parallel Garbage Collection

Contemporary hardware provides abundant parallel processing units that can reduce the program execution time. In a concurrent garbage collection, threads execute with the mutator threads in the same time. However, parallel garbage collection is referred to the Stop-The-world garbage collection, where mutator threads must halt to collect the heap. Parallel algorithms and techniques are widely adopted and tracing garbage collection policies employ parallelism to reduce pause time overhead. Essentially, the collection work must show that there is enough work to be undertaken by multiple cores and that work is divisible between the collector threads. This research considers a stop-the-world collector, in which the collection work includes four main tasks:

1. Enumerating the root set, which is a group of root references, e.g. static fields, threads stacks, and globals
2. Scanning and tracing the root set to discover potential live objects
3. Processing the live object set
4. Reclaiming the garbage space

Details of individual task's amount of work are described in Section 3.6.1.

Parallelizing task (1) requires partitioning the potential root areas, where root references are likely to be found. As described in Section 3.6, there are many root areas and each collector thread scans one or more areas. The copying collector, in particular, divides the card table, which records inter-generational references, into a number of chunks equal to the number of garbage collection threads. The resulting scanned references constitute the root set. Task (2) works on identifying live objects, which need to remain in the heap. Every root reference may form a sub-graph of reachable objects. In this task, the garbage collector traces the transitive closure of each root reference. Since task (1) is expected to generate many roots, parallelism in task (2) is straightforward such that each garbage collector thread traces a number of roots and their reachable object set. Once live objects have been identified, the garbage collector in task (3) processes them in parallel. The copying collector copies the live object set to a different space and the compaction collector compacts them to one side of the space.

The reference graph may have references that are shared between multiple sub-graphs; hence, caution is needed to avoid parallel threads re-processing those objects. The copying collector combines tasks (1) and (2), i.e. as a garbage collector thread discovers a live object, it copies/promotes the live object to the target space immediately. In contrast, the compaction collector requires multiple passes to process the live objects. Task (4) refers to the sweeping or compaction phases. After processing the live object set, the garbage space becomes ready for reclamation.

Parallel garbage collection algorithms encounter common parallelism issues, which could minimize the parallel hardware utilization gains. For instance, garbage collection threads need to synchronize on the root area tasks, i.e. task (1), which may lead to lock contention. In addition, the transitive closure size of each root is likely to be different; thus, load balancing is required to utilize the parallel hardware effectively.

The driving goal of many parallel collectors is to keep the processors busy processing the workload. Few collectors consider the memory implications when designing the parallel collection algorithms. In the next section, we will present processor-oriented and memory-oriented parallel garbage collection algorithms.

Processor-oriented Parallel Collection

One of the earliest works on parallel garbage collection for Java is Flood et al. [2001] collector. Various techniques have been incorporated in their parallel collector. Initially, the parallel phase is preceded by a *static task partitioning* stage, task (1), where an augmented number of tasks are prepared to enable the parallel phase to start with. These tasks are added to a shared queue in order to distribute them across the garbage collection threads. However,

this conventional parallel technique suits the systems with uniform memory access architecture. In NUMA architecture, garbage collection threads compete on the shared queue to acquire tasks, which may direct threads to process remote memory. Additional time overhead is expected when tasks are distributed without considering the memory locality.

When a garbage collection thread runs out of work, it *peeks* on other busy threads and *steals* a reference. Work stealing aims at balancing the load over the collector threads. A thread chooses two random queues and compares their sizes, then takes a reference from the larger queue. Work stealing is facilitated with the Arora double ended queue [Arora et al., 2001]. A thread uses one queue end to push and pop references and the stealing threads pop references from the other end. All queue operations are performed in a thread-safe manner. Since the work stealing algorithm randomly selects the queues, a non-local queue could be chosen.

Endo et al. [1997] propose a different mechanism to balance the load over marking threads. They create *a stealable mark queue* for each thread and wrap it with a lock to synchronize the access. A marker thread pushes references into its local queue and periodically checks its stealable queue size, and if it is empty, the thread gives up all non-local reference to the stealable queue. Once the local queue becomes empty, the marker thread acquires references from its stealable queue, and if it is empty, it steals half of the references of another pending stealable queue.

Instead of using work stealing for load balancing, Wu and Li [2007] implement a task pushing algorithm for parallel marking collection. Collector threads trace the reference graph and push un-processed references into local marking stacks. Occasionally, each thread gives up part of its references and pushes them into non-local queues. They choose to implement N x N single producer/single consumer queues to accommodate pushed references, where N is the total number of collector threads. Idle threads circulate over appropriate non-local queues and acquire references. For instance, when thread *i* runs out of work, it searches for references available in queue *i* of another garbage collection thread *j* , i.e. [j,i].

The task pushing design in this work aims for avoiding synchronization overhead on shared queues. Every thread gets its local queue and $N-1$ non local queues. However, the memory footprint of this design is proportional to the number of collector threads. In fact, as the number of cores increases, we would expect a large space overhead to run the task pushing algorithm. This dissertation proposes implementing a double-ended queue per NUMA node, which is far less than per-thread queue.

Iyengar et al. [2012] study the scalability of the marking phase of the C4 algorithm [Tene et al., 2011]. C4 is the continuously concurrent compacting collector, which is a concurrent mark-compact collector. They report that the duty cycles of the marking phase get worse as the number of threads increases. A primary source of this problem is the contention of work sharing in marking tasks, where multiple threads attempt to atomically update words in a side

bitmap. They modify the parallel marking algorithm of the mark-compact collection policy to reduce the bitmap update contention overhead. The main idea is to split the bitmap into N multiple sections and every section is processed by a single thread. The implementation involves creating N queues for each thread participating in the marking work such that a thread is able to push references to any queue but drain its own queue only. Every marking thread participates in the reference graph traversal and distributes references according to their bitmap section into the corresponding queue.

In the context of NUMA architecture, every bit in the bitmap maps to a word in the heap; thus, the bitmap size is relatively small to fit in modern large cache sizes. In addition, the cache line is big enough to accommodate multiple words; thus, false sharing would occur with high probability. Furthermore, the JVM allocates the bitmap data structure at the initialization phase and it is likely to reside in a single NUMA node. Therefore, the collector threads would saturate the bandwidth of that node.

Memory-oriented Parallel Collection

We have shown that processor-oriented parallel techniques for garbage collection attempt to generate many units of work that are amenable for task working and task stealing. However, memory locality is not generally taken into account in such algorithms. This section surveys memory-oriented parallel techniques to improve the garbage collection locality.

Shuf et al. [2002b] implement a locality-based traversal algorithm. They devise a type-affinity allocation scheme in which related objects of prolific (frequently instantiated) types are co-allocated into clusters of parents and children. At collection time, the garbage collector threads exploit inherent locality of prolific types, and trace reachable object sets, which are likely to be close to each other. They split the heap into several chunks and process local objects in each chunk. References to remote objects are pushed into a shared queue to be processed later on, possibly by other threads.

In their work, Shuf et al. take various implementation decisions to improve locality. For instance, the chunk size is set to be equivalent to the TLB buffer size in order to minimize page misses. In addition, garbage collection threads select roots in top-most stack frames because they are likely to be in cache. For copying collectors, this locality-based traversal algorithm reduces the pause time by 10%. However, for non-copying collectors, the algorithm does not impact the garbage collection performance.

Chicha and Watt [2006] in similar work divide the heap into regions and create a trace queue for each region. The garbage collection threads process local objects only in each region. In contrast to Shuf et al. [2002b], remote objects are pushed into the trace queue of the appropriate region. This mechanism improves the cache locality; however, they consider a sequential case where there is no issue for load balancing.

Oancea et al. [2009] change the work granularity of parallel garbage collection threads to heap partition level. They create a work list for each heap partition to store local objects and a queue for each processor to hold off-partition objects that were discovered in each region. In this way, a heap partition is owned by a single garbage collection thread at a time, which obviates the need for queue synchronization. Since the heap partitions are not mapped to NUMA topology, a collector thread may own a remote heap partition and increase off-node traffic.

Zhou and Demsky [2012] implement master/slave architecture for parallel mark-compact garbage collection. The master core manages the collection phases and distribute memory to individual cores for allocation. At collection time, every core collects its local heap independently. Whenever there is a reference to remote object, the owner core sends a mark message to the master core. The master core forwards the message to the appropriate core, which adds the reference to its marking queue.

One of the key design decisions here is that the hardware, a Tilera processor, uses lightweight messaging exchange support between cores. This feature minimizes queuing operations and synchronizations as compared to Shuf et al. [2002b], Chicha and Watt [2006], and Oancea et al. [2009].

Previous work discussed in this section does not study memory-oriented garbage collection for NUMA architectures. In a recent work by Gidra et al. [2015], the heap is partitioned and every partition is mapped to a NUMA node, similar to the Tikir and Hollingsworth [2005] and Ogasawara [2009] heap layout. A garbage collection thread is allowed to collect its local memory only. If a garbage collection thread encounters a reference to a remote object, it sends a message to the appropriate NUMA node to process that object. In contrast to Zhou and Demsky [2012] collector, the communication infrastructure contains a *software* channel between each pair of nodes. For work stealing, idle garbage collection threads steal work from any node, which could improve memory allocation imbalance between NUMA nodes. Section 2.4 discusses NUMA aware garbage collection in detail.

2.3 Heap Partitioning

The last section presented memory-oriented parallel garbage collection techniques, which generally depend on partitioning the heap. There are many criteria to partition the heap. Objects possess various characteristics which make them distinguishable and amenable to different collection policies. For example, objects live for a spectrum of lifetimes, construct different connectivity patterns, or occupy a wide range of memory sizes. Memory management often produces benefits when discriminating objects and applying different collection mechanisms on them [Blackburn et al., 2002, Jones et al., 2011]. These benefits include

reduced pause time, lower space overhead, and better locality. In this section, various heap partitioning schemes that improve garbage collection locality will be presented.

2.3.1 Thread-local Heaps

One way to partition the heap is to split the heap into two partitions: one partition accommodates objects that are accessible by many threads and the second partition contains objects that have exclusive access by the thread creating them. Many techniques have been employed to classify objects according to their accessibility. The allocator executes specialized allocation methods to allocate objects in a thread-local *heaplet* for single thread access or in a global shared heap for multi-thread accesses.

This partitioning scheme enhances object locality by grouping objects that are accessible by a thread in one heap section; hence better memory page and cache locality in return. In addition, it enables the collector to pause an individual thread and collect the thread's heaplet without interrupting and pausing other threads. In fact, this scheme is proposed originally to reduce the cost of synchronization between mutator and collector threads [Jones and King, 2005]. Garbage collection requires this synchronization for the entire collection time, in stop-the-world collection, where all mutator threads must halt, and for a particular phase in concurrent collection, which pauses the threads to scan the roots. As a result, allocating singly-accessed objects in the heaplets improves locality of the mutator and the collector threads, reduces synchronization and pauses costs as well.

Functional languages, for example Haskell, have the ability to distinguish, ahead of time, between mutable and immutable objects. In addition, functional languages semantics enable the runtime system to duplicate immutable objects. Researchers exploit these features to allocate immutable and mutable objects into different heap sections and apply different garbage collection policies to each section. However, these techniques entail many challenges and several optimizations.

Doligez and Leroy [1993] design a concurrent generational garbage collector for ML type system. The heap layout consists of two generations: the young, which corresponds to the multiple thread-local heaplets and contain immutable objects, and the global, which contains mutable and survived immutable objects. Copying collection is applied to the heaplets to move live objects to the global heap and the global heap runs mark-sweep collection mechanism.

This memory manager strictly prohibits pointers from the global heap to the heaplets or between the heaplets. When a pointer is made pointing to an object in the heaplet, the collector clones the object and places it in the global heap. In addition to copying the object, the collector copies the transitive closure that descends from the object. Object(s) in the

heaplets will have the address of the new copies to avoid copying them in the next minor collection.

Anderson [2010] partitions the heap similar to the Doligez and Leroy [1993] heap layout and applies the same collection policies. Moreover, the memory manager forbids pointers from the global heap to the thread-local heaplets and between heaplets to allow independent and asynchronous heaplet collection. Although the Haskell language is able to determine object mutability ahead of time, Anderson observes that mutable objects are subject to mutation for short time then become immutable. In contrast to Doligez's heap layout, Anderson allocates mutable and immutable objects in the heaplets. However, he tackles the problem of pointers between the heap sections in a different way. When a pointer to an object in a heaplet is written to an object in the global heap, a write barrier is called to collect that heaplet and copy live objects to the global heap. In this case, the pointer will point to an object in the global heap.

Since access to mutable objects are expected, the number of heaplet collections would increase. Two optimizations have been proposed to overcome this overhead. First, Anderson analyses the root cause of the minor collection time overhead and found that the *stackwalking* work constitutes largely to the collection time. The stackwalking refers to traversing the frames on the execution stack. Therefore, stackwalking optimization, for example data caching, was able to reduce the overhead by 50%. Second, intuitively, the existence of mutable objects in the heaplets will frequently initiate garbage collection; thus the write barrier clones and allocate objects in the global heap. In contrast to Doligez and Leroy, Anderson attempts to optimize the cost of global heap space consumption, which is caused by the cloned objects, and increases the number of major collections.

Mutable objects tend to be more common in object-oriented languages than functional languages. The ability to split objects according to their mutability and the management of heaps' cross-section pointers may require different techniques.

Steensgaard [2000] designs the heap layout to have generational partitions. The young and old generations contain per-thread heaplet as well as a shared heap for shared objects. All partitions are collected with a copying collector. The minor collection moves live objects, *whether private or shared*, to the old generation. Nonetheless, *all* heap partitions are collected in the same time. This limitation enforces the mutators to rendezvous and a single thread copies live objects in the shared heap only to the old generation. After that, all threads collect their heaplets concurrently and resume normal execution independently. Although heaplets are collected concurrently, latency in Steensgaard's collector does not benefit from this parallelism since the shared heap is collected as well. Major collection works similarly as the minor collection but objects are copied to a new section in the old generation.

The collector performs static escape analysis to distinguish between objects that have sole

accessors and shared objects. Escape analysis results enable specialized allocation methods to create objects in the appropriate location, whether in the heaplets or in the shared heap. However, static analysis imposes several drawbacks when classifying objects as shared or private. On one hand, the decision is based on the allocation site; all objects created by this site are considered shared whenever it contains a shared objects, no matter how many objects are private. On the other hand, if an object lives long as private, and then becomes shared, the analysis will treat it as a shared object till its death.

In contrast to the static analysis technique, Domani et al. [2002] determine that, at runtime, the status of an object whether it is shared or private using write barriers. Object locality state is indicated by a bit stored in a bitmap. Objects are initially allocated in local heaplets except objects that are global by nature, for example Class objects; thus they are allocated in the shared heap. When an update to a pointer is required, a write barrier is invoked to check if a private object becomes a descendant of a shared object. A write or read barrier is a mechanism for the garbage collector to execute memory management code when a read or write to an object occurs. If this is true, the private object and its descendant objects are marked as shared objects in the bitmap. The collector applies a mark-sweep algorithm to collect the heaplets and the shared heap and the whole heap is compacted whenever there is not enough space. A thread performs a minor collection independently from other threads on local objects only when the heaplet is full. Major collection is initiated if the shared heap is full or when the heaplets cannot satisfy local allocation requests.

A major drawback of Domani et al. is that the write barrier does extensive work in traversing and marking globally accessible objects and their descending objects. Jones and King [2005] avoid the write barrier overhead by taking a snapshot of the heap at runtime and incorporate static analysis. Initially, objects are allocated in the shared heap. At certain point in the execution order, the runtime system takes a snapshot of the classes loaded up to that point. After that, the snapshot is statically analysed and objects are classified as strictly local, optimistically local, and global. According to the analysis results, a number of specialized versions of methods are generated and fed to the JIT compiler and to the interpreter to allocate objects into appropriate heap sections. Pointers are forbidden from global objects to the heaplets or between the heaplets; therefore, every thread collects its own heaplet independently from other threads.

Marlow and Peyton Jones [2011] study the effect of promoting private objects to the global heap. The results show that promoting the transitive closure of a pointer written to a shared object incurs high cost. Consequently, instead of promoting the transitive closure of an object, the memory manager accepts pointers from the shared heap to the private heaplets and protects private object accesses by read barriers rather than write barriers. Whenever an access to a private object is requested, the read barrier checks if the private object is owned by the thread itself then it grants an access to the object directly. Otherwise, the read

barrier works as a guard asking the owner thread to move the object to the global heap. The heap layout then consists of per-thread heaplets each with two parts. The first part acts as a traditional nursery with a copying collector. The second part is called the *sticky heap*, which contains objects that lack read barriers, hence are private and immovable and this part is collected with mark-sweep collector. The shared heap is collected with a stop-the-world collector.

Cohen et al. [2006] attempt to cluster the sub-heaps to reduce the number of thread-local heaplets and the number of objects in the shared heap. The main idea behind this work is that a number of threads share a sub-heap where allocated objects are accessed only by those threads. At collection time, only these threads are suspended while other threads remain executing. They use a clustering software module [Mancoridis et al., 1999, Harman et al., 2005] technique, which is based on a hill climbing algorithm to find optimum heap clusters.

The clustering technique works by creating a *Thread Dependency Graph* (TDG), a directed graph that represents threads as nodes and dependencies between threads as edges. The dependency indicates threads' accesses to an object. For instance, if object *O* is accessed by Thread *A* and *B*, then a dependency is created between threads A and B. The graph is built using system traces of previous runs and fed into parallel hill climbs. The results provide the proposed heap layout which consists of a number of sub-heaps where each sub-heap is mapped to one or more threads and a shared heap to accommodate objects that break the heap clusters. Experiments show that heap clustering reduces the number of sub-heaps and the total number of shared objects in the shared heap.

2.4 NUMA Heaps

The heap partitioning schemes described in the last section focus on various object characteristics. This section surveys a physical memory heap partitioning criterion, the NUMA heap. Heaps in modern machines are physically partitioned between NUMA nodes. Therefore, object placement techniques for NUMA heaps have gained attention recently due to the growing availability of NUMA machines. Moving objects between NUMA nodes can potentially improve mutator threads performance by placing objects frequently accessed by threads of a NUMA node into the same node.

Tikir and Hollingsworth [2005] study the impact of applying dynamic page placement techniques, which are used for applications with regular memory access patterns, on Java applications. They examine three placement policies on the SPECjbb2000 Java application. First, static-optimal: a technique that has the access information of each heap allocation. Objects are placed in a memory page local to the processor accessing them. Second, prior-knowledge: this technique knows the access information about surviving objects and mi-

grates them to memory pages local to the processor accessing them the most at garbage collection time. Third, object-migration: measures the access frequency for each object since the start of execution and the garbage collector uses this information to migrate objects to the processor's local memory pages.

The study results show that the prior-knowledge policy provides the best results in both the young and the old generation. In addition, the object-migration technique reduces the non-local memory accesses in the old generation. Therefore, the authors suggest to partition the heap according to the system's NUMA topology. The heap will have the following layout: the Eden space of the young generation and the old generation are segregated into a number of segments equal to the number of NUMA nodes. They did not partition the survivor space because the experiments showed low memory accesses to the survivors; hence low potential benefit from partitioning this space. This heap layout is implemented and evaluated using a simulator. Calculating the memory access frequencies is the crucial component of this research; however, the values were obtained from previous runs of the workload and fed to the simulator in advance. The results show a reduction of non-local memory accesses by 40% compared to the original heap layout in the Hotspot JVM.

Ogasawara [2009] criticizes the method used to calculate the memory access information, which was based on trace file processing. The inaccuracy of matching memory access events with objects and the time consumed by the garbage collector to find out the *preferred location* of an object in which it will be moved to; drive the researcher to consider an easier and lower overhead technique to calculate the preferred object location. He employs heuristic information to determine the preferred object location and calls this the dominant-thread (DoT) information. Heuristics include the thread identifier that acquires the object lock or reserves the object. This information is available in the object header and getting it incurs very low overhead. In case an object does not hold this information, the object gets the preferred location calculated for the object referencing it directly or indirectly.

Moreover, the heap layout is similar to Tikir's heap, however, Ogasawara partitions the survivor space as well. The old generation consists of a number of segments matching the number of NUMA nodes. The Eden space in the young generation consists of multiple segment groups. Each group contains multiple segments to reflect the NUMA topology. In addition, the survivor space contains one segment group only. Mutators request memory from the corresponding segment in the Eden space. The allocation policy is not strict so it can extend the memory from the next segment as needed.

Garbage collector threads identify the preferred location of a survivor object using the dominant thread information. First, the preferred location of objects directly pointed at from the thread stacks is the NUMA node of the thread running on it and, which can be retrieved by system calls. Second, for objects that are locked or reserved, the preferred location is the

thread identifier stored in the object header. Third, other objects use their parent's preferred location. The garbage collector moves the live object set to the preferred locations in the survivor space or the old generation.

Gidra et al. [2013] relate NUMA heap partitioning to the memory allocation policy that provides potential scalability benefits over increased core counts. They study the memory allocation behavior of multi-threaded applications and conclude that, as a common programming practice, the initialization phase is done by a single thread. Given that Linux memory allocation policy for NUMA systems uses *First-Touch* policy [LinuxMemPolicy, 2015], memory pages of the Eden space will be mapped to a single NUMA node; causing future allocation requests to be satisfied from a single node, which saturates the memory bandwidth and the cross-chip interconnection link of that node. Accordingly, they suggest to interleave memory pages of the Eden space to avoid memory imbalance issue, however, interleaved memory policy ruins memory locality because objects will be scattered across all NUMA nodes. In addition, the young generation accommodates short-lived objects that are mainly accessed by the thread creating them; hence, fragmenting the young generation in correspondence to the NUMA topology, similar to Tikir and Ogasawara studies, is likely to improve the mutator and collector threads locality.

Another observation is that long-lived objects in the old generation benefit much from memory balanced allocation policy. When the garbage collector uses the interleaved memory policy, survivor objects would be distributed across the NUMA nodes, providing better memory bandwidth of the memory controllers and the off-chip interconnect links. Gidra's NUMA-Aware Parallel Scavenge (NAPS), a stop-the-world throughput-oriented garbage collector in the Hotspot JVM, employs fragmented space in the young generation and interleaved memory policy in the old generation and experimental results show that stop-the-world collector is able to scale well.

2.5 Object Locality

Object oriented programming languages, for example Java, make extensive use of dynamically allocated heap objects. The memory manager allocates an object in the heap and returns a reference to it. Any access to that object is through its reference. This kind of memory allocation gives the memory manager the ability and flexibility to allocate and relocate objects in any space within the runtime heap boundaries. The memory manager needs only to ensure that the object's reference is up to date since it is the only way to access the object.

The flexibility of object movement in the runtime heap challenges the garbage collector to manage spatial and temporal locality. Therefore, a large body of research works on improving object locality. It spans a wide range of subjects: from placing frequently accessed

objects adjacently in a memory page to placing frequently accessed fields of an object or group of objects in a single cache line and from the reference graph's traversal order to NUMA heap partitioning.

This section reviews three optimization areas for object locality. These optimizations consider different kinds of locality— based on physical, e.g cache lines, and logical, e.g. object, components. First, the object traversal order and its impact on object locality are discussed using static analysis techniques. Second, cache locality is studied using dynamic analysis of object access patterns. Third, several optimization techniques used for multicore and NUMA architectures are presented.

2.5.1 Cache Locality Optimization

Java objects are generally small in size [Bacon et al., 2002, Chilimbi et al., 1999a, Chilimbi and Larus, 1998]. A cache line (usually 64 bytes in size) can accommodate multiple objects. This feature attracts researchers to explore possible techniques to improve object temporal and spatial locality at the hardware cache level. Obviously, frequently accessed objects are potential candidates to live in the same cache line. In fact, literature goes further and explores which fields of an object have higher access rate than the others. Object's fields can be reorganized such that "hot" fields placed next to each other. Consequently, frequently accessed fields of an object placed together in the same cache line. A step further in the research enables hot fields of multiple and *different* objects to reside in a single cache line.

The main challenge to the cache locality optimization is how to identify hot fields of an object, so that they can be co-located in the same cache line. This section reviews a number of cache locality optimization approaches. A fundamental technique in these approaches is that they profile the program at runtime to record data access information and calculate the hot fields.

Chilimbi and Larus [1998] develop a graph-based technique to identify hot objects and create cache-conscious data structures. At program execution, a data profiling system records the object's base address for each load operation and enters it in an object access buffer. The garbage collector uses the data profile buffers to construct a weighted undirected object affinity graph. Nodes in the graph encode objects and edges encode temporal affinity. At collection time, the collector uses the affinity graph to layout objects with high temporal affinity next to each other. The outcome of this technique is high spatial and temporal cache locality since frequently accessed objects would reside closely and the same cache line is going to be used soon.

Nonetheless, the average overhead of runtime profiling constitutes up to 6% of a Cecil program's execution time. Since the object affinity graph is constructed at every collection, they

apply this technique on the old generation only because minor collections are triggered more often which may cause significant overhead.

Calder et al. [1998] implement a different cache-conscious data placement technique. They use a compiler-directed mechanism that assigns addresses for global variables, stacks, heap objects, and constants to reduce data cache misses. Their technique requires a training run to gather data access information. The collected information is fed to the compiler to map proposed new virtual addresses for local and global variables and constants. For heap objects, they modify the memory allocator to assign the new addresses at runtime. The results show substantial locality improvement for global variables and stack objects; however, heap objects obtain insignificant performance improvement.

Novark et al. [2006] present an approach that enables programmers to change and control the object layout at collection time. Programmers annotate the code and provide a custom object layout for a class, which works as a hint to the runtime system to arrange objects in memory. At garbage collection time, the collector invokes the custom object layout methods to place objects into contiguous memory. Results show that a custom object layout reduces cache misses by 50%.

For non-garbage collected environments, Chilimbi et al. [1999b] propose two techniques for data reorganization: clustering and coloring. Clustering attempts to group data structure elements that have temporal affinity in a cache line. They target tree-like data structure and develop a tool to reorganize the data structure elements into sub-trees that are laid out linearly. On the other hand, the coloring technique organizes data in the cache to avoid resource conflicts. A cache has limited number of concurrently accessed data elements in a cache block. Thus, coloring maps concurrently accessed elements to non-conflicting regions of the cache to reduce cache conflict misses. However, these techniques require programmer's intervention to select related objects. In addition, they target L2 cache to get larger cache size and put many objects in the same cache line. When the cache line becomes full and other objects cannot fit into it, the authors attempt to co-locate objects into the same virtual memory page; hence, TLB misses will be minimal. The programmer has to intervene here and add hints that these objects are likely to be accessed together.

The memory hierarchy and the large data structures divert the research on optimizing cache locality to explore fine-grained techniques. Object fields often have different access frequencies. Instead of wasting a cache line with rarely accessed fields, only frequently accessed fields of objects should reside in the cache line.

Chilimbi et al. [1999a] suggest *structure splitting* to arrange internal organization of structure instances. The main idea of structure splitting is that Java classes have different access frequencies and that enables the class to be divided into hot (frequently accessed) and cold (rarely accessed) portions based on field access profiling. This technique requires static

analysis to provide class information and dynamic analysis to measure class instantiation and access statistics. Profile data are given to the compiler to generate code with structure splitting optimization. Class splitting involves injecting a pointer from the hot class to the cold class. Accordingly, hot class construction must create cold class instance first. The results showed around 20% performance improvement.

Furthermore, internal organization of large structures that span multiple cache lines can also be reorganized. Fields in a structure are ordered logically, which do not necessarily correspond to the temporal access patterns. Consequently, this logical layout may incur unnecessary cache misses. Moreover, a few fields of each class instance are accessed by the most active parts of the code [Truong et al., 1998]. Therefore, laying out these fields that are often referenced together into the same cache line improves the program performance [Panda et al., 1997].

As in class splitting, Chilimbi et al. [1999a] employ static and dynamic analysis to construct a field affinity graph of a structure and generate recommendations for field reorganization. Fields with high temporal affinity are clustered in the same cache line. However, such class splitting and field reordering techniques require substantial programmer effort.

The internal field reorganization of a class may still fill the cache line with unnecessary cold fields. Truong et al. [1998] suggest filling the cache line with frequently accessed fields of different instances of a class and call this technique *instance interleaving*. This is based on an observation that the reference pattern of a program often accesses a few fields in each instance and these fields are not enough to fill a cache line. In addition, their technique ensures that when interleaving many instances, these fields are likely to be contemporaneously accessed, therefore, fields are mapped to different cache sets to eliminate conflict misses.

2.5.2 Memory Page Locality Optimization

A heap-allocated object is accessed through a pointer or a sequence of pointers. This form of reachability does not imply locality, in general. Objects that have similar access patterns may be allocated in distant memory locations. In addition, objects that survive a garbage collection may change location. Various criteria can be used to improve object locality in virtual memory. For instance, allocating objects that have been referenced together or objects with high access frequency next to each other may obtain high spatial locality.

Static Object Reordering

One way to improve object locality in the virtual address space is the traversal order. The garbage collector treats live objects that the application still accesses as a reference graph. A

traversal order may co-locate parent and children objects together or split them away. There are many traversal orders discussed in the literature. **Breadth-first** is a common traversal order and it slices the graph horizontally keeping nodes in the same level close to each other. This traversal is simple and cheap using a few pointers that resembles a queue. Objects to be processed are dequeued from the head of the queue, while their descendants are enqueued into the tail of the queue. Cheney's algorithm [Cheney, 1970b] employs breadth-first order in his collector to copy objects between semi-space heap regions. A major effect of this traversal is that parental objects are separated from their children. In fact, they could be allocated in different memory pages, perhaps in different NUMA nodes. Objects in typical data structures, e.g. linked lists, are likely to be referenced together; thus, breadth-first would not satisfy optimal object locality for such data structures.

As opposed to the breadth-first order, **depth-first** traversal places parent and children objects together. Sequential access to objects may benefit from this order since objects will be next to each other. Many collectors involve depth-first traversal, e.g. [Courts, 1988, Moon, 1984, Stamos, 1984]. Although depth-first traversal keeps elements of lists close to each other, Courts [1988] reports only 10%-15% performance gains. The reason behind this low percentage was that the locality improvement targets system images, i.e. a copy of a system utilities and libraries, and lists were not the common data structure incorporated in a system image [Lam et al., 1992].

Wilson et al. [1991] argues that most data structures are tree-like and neither breadth-first nor depth-first provide optimal locality. They implement a hierarchical decomposition traversal algorithm and hypothesize that tree structures should be best grouped in sub-trees. This technique will group together a node with its closest descendants, which is better than depth-first which covers only one branch. However, this technique was tested on a program with various data structures and the results did not provide better locality. They conclude that a fixed traversal may not yield optimal ordering.

In a different, but related problem, the system image is mainly organized as library functions, typically including compiler, browser, and editor [Andre, 1986]. These functions remain live throughout execution of every program. To achieve better locality, functions should be grouped according to a correlation aspect. One aspect to organize the library functions is the creation order, which is the time functions are presented to the compiler. Another aspect is to group functions according to the transitive call sequence [Lam et al., 1992]. Both techniques gain better locality and fewer page faults.

Different object ordering can be combined to provide adaptive ordering according to various object correlation factors. One way to implement adaptive ordering is to adapt the traversal to the object type [Lam et al., 1992]. In this technique, the collector checks the object type and applies an appropriate order to improve the locality of that object. For instance, objects

of list type are processed in a depth-first order and objects of tree type are traversed in a hierarchical decomposition order.

All analysis techniques considered so far in this section are static, i.e. object layout strategy is set ahead of time. Static analysis provides information about how objects can be ordered not how objects will be accessed [Courts, 1988]. Programs often exhibit different execution phases, in which access to objects changes from one phase to another. Static analysis and offline profiling may provide misleading information or do not capture phase changes. A dynamic (online) profiling would provide accurate measures.

Dynamic Object Reordering

Runtime systems may include Just-In-Time (JIT) compiler for fast code execution compared to code interpreter. In contrast to the classic compiler, JIT compiler compiles the code at runtime, and it usually compiles hot methods, which are frequently executed methods. Huang et al. [2004] utilize method sampling, which is used by the adaptive JIT compiler to optimize hot methods, to identify hot fields in the hot methods. The overhead of this profiling is less than 2% since it piggybacks on the system's method sampling. At garbage collection time, the collector copies referents of the hot field with their parent. Furthermore, they improve their online object reordering technique by changing the hotness threshold to respond to phase changes.

Chen et al. [2006] combine cache and page locality optimizations in the same system. They instrument the JIT compiler of the Common Language Runtime (CLR) to record object accesses in a buffer and insert monitoring code to gather certain metrics that guide locality optimization. For cache locality, they use Chilimbi and Larus [1998] technique, which constructs an affinity graph to co-locate related objects in the same cache line. Objects that are not moved during cache locality optimization and still have frequent accesses are grouped in separate pages of the heap. One of the most advantageous contributions in their work is that the locality optimization is decoupled from the normal garbage collector, which is a reaction to the heap space constraints. Whenever the program's data access pattern changes due to program phase behaviour, the system triggers the garbage collection to respond to the data locality changes. This proactive calling to the garbage collection is managed by the object allocation rate, which is a reliable indicator of locality phase change. The results showed 17% improvement in the program performance.

Guyer and McKinley [2004] develop a dynamic object co-location algorithm to group connected objects in *the same heap space* of a generational collector. One advantage of this work is to eliminate cross-generational pointers, which incurs write barrier overhead. To discover potential connectivity between objects, the algorithm employs static analysis to find old objects that will reference new objects. At runtime, a new allocation routine puts newly

allocated objects in the same region as the old object. Co-locating dynamically connected objects improves the garbage collection time by 50%, and the total execution time by 10%.

Wimmer and Mössenböck [2006] collect access profiles using read barriers inserted in the machine code by the JIT compiler. For each class, whenever a field load instruction is executed, a counter that tracks this field is incremented. As the program executes, fields that reach a load threshold are considered hot and added to a hot field table. This table is used by the garbage collector to co-locate parent and child objects at collection time.

Another aspect of grouping related objects is the notion of prolific types. Shuf et al. [2002a] observe that some object types frequently instantiate many objects and those objects tend to live for short time. They suggest to partition the heap into two regions: one region for prolific type objects and the other region for non-prolific objects, which are analogous to the young and old generation of a conventional generational collection. The garbage collection performs *minor* collections in the prolific region; however, survivor objects remain in the same region and no object is promoted to the non-prolific region.

Yu et al. [2008] attempt to improve the identification method of prolific types at runtime. They create two spaces: reusable and non-reusable space to co-locate prolific types. The main difference to Shuf et. al is they locate prolific objects of the same type side-by-side in the same memory block. When collecting the reusable space, the prolific objects in a memory block are likely to be dead, hence the memory block is recycled. In contrast to Shuf et. al, both spaces are collected every time.

Object Inlining

Objects in object-oriented languages often contain fields that point to other objects. This kind of connection incurs field load overhead to access the referenced objects. Object inlining is an optimization that embeds referenced objects into their referencing objects. The main advantages of object inlining are to allocate objects in a consecutive memory addresses in the heap and to substitute the reference in the parent object with address arithmetic. Accordingly, object inlining optimization improves cache and page locality by co-locating objects next to each other.

Typically, object inlining is performed by a static compiler; where static analysis and transformations techniques pass through the code to spot possible inlining opportunities. Whenever the compiler finds candidate inlinable objects, it replaces the allocation sites of these objects with one site, which allocates inlined object. Research that explores this optimization includes the work of [Dolby and Chien, 2000, Laud, 2001, Lhotàk and Hendren, 2005].

Instead of relying on the static compiler, Java provides a JIT compiler to dynamically compile the bytecode into the machine code. This feature can improve the inlining decisions

taken at compile time by inlining objects that are frequently accessed. In addition, since the garbage collector moves objects around the heap, it can support object inlining optimization by placing inlinable objects consecutively in the heap.

Wimmer and Mössenböck [2007] implement an automatic feedback-directed object inlining optimization. Their algorithm involves a runtime monitoring system, which injects read barriers to count field accesses and detect hot fields. The generational garbage collector uses the profile data to find hot fields and co-locate the parent object and the child object that are connected by the hot field in consecutive memory space in the young generation. Moreover, when the parent object is eligible for promotion to the old generation, the garbage collector moves the parent and the child objects together, which is considered as a form of object pretenuring to the old generation.

Veldema et al. [2005] generalize object inlining optimization and suggest combining related objects together if they are: eligible for object inlining, have similar access pattern, or if they have similar life spans. The solution applies several static analysis techniques to find objects where combining them might be beneficial. The results show reduction in the total execution time by up to 34%.

2.6 Object Clustering

In a larger memory region, i.e. many memory pages, objects may exhibit certain characteristics that can be utilized to improve garbage collection performance. For example, a group of objects may have similar lifetimes or a particular object type could allocate objects with high proliferation frequency. Such properties enable the garbage collector to apply different optimizations on each object group.

Generational garbage collectors exploit the weak generational hypothesis [Stefanović et al., 1999, Ungar, 1984], which states that most objects die young. Consequently, an age-based object segregation scheme places an object according to its lifetime. The young generation, which accommodates short lived objects that are often accessed frequently, can be collected independently from other partitions. Collecting the young generation frequently would reduce pause time and the overall amount of work. Moreover, reclaimed memory space from collecting the young generation is large (most objects die young) and that enables sufficient memory space for future memory allocations.

However, the old generation does not exhibit the same properties as the young generation [Hayes, 1991]. The space occupied by old objects is larger than for the young generation; hence, collecting the old generation is time consuming. Several techniques have been proposed to enhance the old generation collection, for example incremental collection [Hudson

and Moss, 1992] and object pretenuring [Ungar and Jackson, 1992, Blackburn et al., 2001, Singer et al., 2007].

The connectivity between objects is another criterion for clustering objects. Both kinds of connectivity: *direct*, where object A points to object B or *transitive*, where object B is reachable from object A can reveal object grouping criteria. A study by Hirzel et al. [2002] examines different connectivity patterns and object lifetime and deathtime. They conclude that connected objects that are reachable only from the stack are shortlived; whereas, objects that are reachable from globals live for long time, perhaps immortally. In addition, objects that are connected by pointers die at the same time.

This object connectivity behavior can be utilized to improve the garbage collection performance. The same authors, Hirzel et al. [2003] segregate the heap into many partitions, each contains a set of connected objects. They use compiler analysis to determine the connectivity of objects and eliminate write barriers by avoiding pointers between partitions. Since connected objects usually die together, the garbage collector chooses some partitions where much space can be reclaimed. To trigger the garbage collector, an estimator is used to annotate the partitions with a high proportion of garbage to indicate the need for collection. They developed a simulator *gcSim* to evaluate their solution and report improved performance over other garbage collection implementations.

Once a garbage collector implements a policy that segregates objects based on certain object properties, it would apply this policy on all objects regardless of the application behavior. The user program usually exhibits different phases, in which the memory requirements are different. The memory manager should recognize and exploit phased behavior. Jones and Ryder [2008] study Java object demographics and find a relationship between allocation sites, for example JIT compiler and the user program allocations, and both the program phase behavior and object lifetime distribution. Objects allocated by these allocation sites cluster strongly and are stable across different inputs. They conclude that allocation sites create objects with consistent behavior; thus the garbage collector works on objects from key allocation sites where objects are expected to die [Jones and Ryder, 2006].

2.7 Data Placement Policies

NUMA architecture increases the space of memory page mapping options. Data can be allocated in a local or remote NUMA node relative to the thread's node that accesses it. In contrast to UMA architecture, data location may impact the access latency; hence, the overall application performance.

Managed runtime systems reclaim garbage memory automatically and this implies a high likelihood of object movement between NUMA nodes for generational heaps. In fact, an

object may live in multiple NUMA nodes during the course of its life. A memory placement policy determines an initial object location for the application threads; however, the garbage collector changes the object location and alters the memory placement policy. Furthermore, programs exhibit different execution phases where data access characteristics at each phase may be different; hence, a static data placement policy may be suboptimal.

Existing deployed data placement strategies consider two factors to provide optimal NUMA machine performance. Firstly, threads and data are placed in the same NUMA node to increase locality and avoid remote access overhead. Secondly, data is distributed across the system's nodes to avoid bandwidth saturation. This section will review various data placement policies.

In the late 1990s, remote access overhead in NUMA systems took 3 to 5 times longer than local access [Verghese et al., 1996]. This overhead was due to the *legacy* wiring techniques which resulted in major delays in the interconnection links between nodes. To overcome this issue, a large body of research attempts to improve locality by placing data in the local node of the core accessing it.

Several techniques have been developed to improve data locality. To avoid the large remote/local access latency ratio, memory pages can freely move between NUMA nodes to enable local access. This technique is called memory page *migration*. A memory page that is frequently accessed by a remote core is migrated to the core's node.

Previous memory page migration policies were developed in the context of non-cache-coherent NUMA systems, for example, Bolosky et al. [1991], LaRowe et al. [1991]. Kernel-based NUMA management policies are modified to explicitly move memory pages in response to page fault events. LaRowe et al. [1991] exploit page fault signals and modify operating system memory management modules to implement a parameterized memory page migration policy. For instance, a shared memory page between NUMA nodes may have temporal access. Thus, the memory page may *bounce* between NUMA nodes and affect access latency. To avoid actively-shared memory page bouncing between NUMA nodes, they set a *freeze-window* to hold memory page migration for a certain period then *defrost* it. They concluded that tunable dynamic memory page migration can have dramatic impact on application performance.

Bolosky et al. [1991] use reference traces from a variety of applications to drive simulations of different NUMA page placement policies. In addition, they employ a cost/benefit model to decide whether the cost of moving a memory page outweighs the cost of remote memory access overhead. Chandra et al. [1994] investigate the effectiveness of using TLB misses as an indicator for memory page migration on cache-coherent NUMA systems. However, they report that there is no improvement on the response time for the workloads due to internal issues of their virtual memory system. Instead, they carried out a trace-driven study

to evaluate the usefulness of using TLB misses to migrate memory pages and found that TLB misses can be used to improve application performance.

Verghese et al. [1996] employ a cache miss heuristic to create a decision tree for memory page migration. Nonetheless, using cache misses captured by sampling or trace-driven techniques to migrate memory pages does not gain any benefits. They report that the main causes were due to the processor synchronization and TLB flushing overhead.

In contrast to high performance applications which gain performance from memory page migration, object-oriented languages create small objects so a memory page can accommodate plenty of them. Therefore, memory page migration might not be beneficial for Java-like programs [Tikir and Hollingsworth, 2005].

Shared data is likely to complicate memory page migration optimization. As described earlier, memory pages may bounce between NUMA nodes if they contain intensive data sharing. Alternatively, memory page replication attempts to duplicate highly shared memory pages and enable different threads to access them.

2.8 Conclusion

This chapter has reviewed the general area of automatic memory management, with particular focus on identifying and exploiting memory locality. The literature survey of the existing research in the field of garbage collection shows that most studies target UMA architectures. A great deal of studies focuses on improving object locality (Section 2.5 and Section 2.3).

NUMA related garbage collection research have emerged in the last decade. Few studies have examined optimizing garbage collection for NUMA architecture. The review of these studies (presented in Section 2.4) shows that changing an object location due to copy/compact collection may impact the application and the garbage collection performance. Therefore, there are various techniques to calculate the new destination of an object. An object can be relocated to the NUMA node of the core accessing it the most ([Tikir and Hollingsworth, 2005]), to the same NUMA node ([Gidra et al., 2013]), or using heuristics to identify the appropriate NUMA node ([Ogasawara, 2009]).

A common mechanism to relocate objects to the appropriate NUMA node in those studies is to partition the heap into segments and map them to the underlying system's NUMA topology. Managing NUMA segments is a non trivial task. Unbalanced memory allocation may frequently fill some segments and increase the rate of garbage collection cycles. In addition, a segment resizing policy may allow different NUMA memory pages to be in the same segment.

Furthermore, garbage collection threads must know the original NUMA node of every object before moving it. The cost of this operation is proportional to the live object set size. This dissertation contributes to the field by investigating alternative techniques to the existing ones by avoiding heap partitioning and minimizing the cost of object location retrieval which could provide better performance gains.

The next chapter presents technical background of hardware and software components that interact with garbage collection and may impact its performance.

CHAPTER

3

TECHNICAL BACKGROUND

This chapter presents a brief technical background on hardware and software components related to my work in memory management. Since the goal of this dissertation is to optimize garbage collection on NUMA architectures, it is important to know how the underlying hardware and software execution stack interacts with memory. NUMA architectures present complex topologies and a hierarchical memory subsystem. By comprehending such an interaction, we will be able to understand and better predict the program's performance.

This chapter discusses three main topics:

1. First, it describes the evolution of parallel architectures from unicore to multicore processors. Since the experimental system of this research is based on AMD Opteron processor, this chapter illustrates the memory components that affect data placement decisions. Information is extracted from AMD developer manuals and various system engineers' articles.

2. Second, it highlights NUMA configurations and tools that Linux provide for memory and thread management.

3. Third, based on the OpenJDK Hotspot JVM source code, this chapter describes the implementation of the *Parallel Scavenge*: a Stop-The-World garbage collection policy.

3.1 Introduction

Advanced technologies for the processor enable the clock rate to scale up until the time it hits the *power wall*, where the heat and power dissipation impede clock frequency speedup [Liu et al., 2009]. An architectural response to this problem has driven the processor architects to integrate multiple cores in a single chip. This new architecture forms the basis for contemporary and prevalent multicore and manycore processors available for a wide range of usage; from end-user mobile devices to large enterprise servers.

Over the past two decades, computer scientists have anticipated the *memory wall* phenomenon [Wulf and McKee, 1995, McKee, 2004], where the memory bandwidth would be unable to support the abundant memory access requests issued by an ever increasing number of cores. This problem occurs as a result of the disparity between processor speed and memory access latency [Hennessy and Patterson, 2011a]. Figure 3.1 depicts a single processor performance in terms of memory requests per second compared with memory access performance per second. Contemporary processors employ sophisticated and hierarchical data communication channels between processing units and memory. Modern applications increasingly utilize many parallel processing units to increase the performance. These parallel processors put pressure on the memory bandwidth to access data but the bandwidth is limited, and its improvement is lagging behind the processor advancement. To address this issue, technologies have been developed to expand memory bandwidth and reduce memory access latency.

Multicore processor's design challenges the shared memory model, where all cores access a shared memory space. It is observed that the memory bandwidth will be under pressure due to the huge number of simultaneous memory access requests from integrated cores. Therefore, the memory design response has shifted the shared memory model from a centralized memory area to a distributed shared memory model. This design implies that memory is distributed across the system while having a shared addressing scheme.

The implication of such a design is that cores can access any memory area in the system. However, memory access latency is different from one memory area to another. Table **??** shows the relative memory access latencies for traffic between these four nodes.

This access latency variation is due to the interconnection delay between distant memory areas. To take advantage of distributed shared memory architectures, operating systems attempt to place data close to the processing units to avoid communication overhead. In addition, they take into account different data placement factors, for example memory balance to distribute data across all the system's memory banks. At more abstract software execution layers, such as the virtual machine, data placement is usually devolved to the operating system. However, program's performance may be suboptimal due to features of the system's topology which are not tolerated by the OS's memory policy. The OS default

Node	1	3	4	6
1	10	16	16	22
3	16	10	16	22
4	16	16	10	16
6	22	22	16	10

Table 3.1: NUMA delay time between nodes.

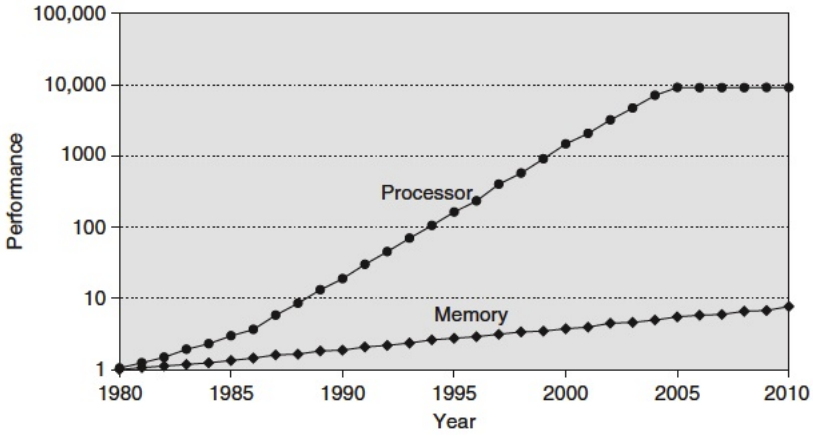


Figure 3.1: The performance gab between processor and access to main memory. Source: [Hennessy and Patterson, 2011a]

memory allocation policy may separate data from processors and increase memory access latency. Therefore, memory allocation policies need to be aware of the underlying hardware to reduce the overhead of remote memory accesses.

This chapter describes related the hardware and software components that affect an application's data placement. Section 3.2 describes parallel architecture evolution. Section 3.3 highlights NUMA architecture and low level hardware specifications. In particular, this section is dedicated to the AMD Opteron processor architecture, which is the platform used for all experiments in this dissertation. An overview of Linux memory allocation policies is provided in Section 3.4, then Section 3.5 discusses the virtual to physical memory page mapping mechanisms. Section 3.6 describes the implementation of garbage collection policies in the OpenJDK Hotsopt JVM. Specifically, it considers the Parallel Scavenge collector: a Stop-The-World garbage collector and reviews implementation details from the source code.

3.2 Parallel Architectures

The current trend to support scalable performance is to augment the number of cores per processor. These cores can be organized in different ways according to their data flow and control flow. Flynn [1972] identifies four categories of parallel architectures:

1. Single-Instruction stream, Single-Data stream (SISD): This architecture consists of a single processing element and can access a program and data storage. For example, unicore processors implement SISD architecture and represent conventional sequential computers following the Von Neumann model.

2. Multiple-Instruction stream, Single-Data stream (MISD): There are multiple processing elements, each executing its own program. However, they have single access to data in the global memory. Processors may execute different instructions but they have identical data as operand. For example, systolic arrays include a network of hard-wired processor nodes to perform a specific operation such as parallel convolution tasks. This type of architecture is very limited and has not been built commercially [Rauber and Rünger, 2010a].

3. Single-Instruction stream, Multiple-Data stream (SIMD): In this architecture, multiple processing elements execute the same instruction stream but different data is loaded from global memory with private access. Vector processors are good examples of this category.

4. Multiple-Instruction stream, Multiple-Data stream (MIMD): Here, multiple processing elements load separate programs and separate data from the global memory. They work asynchronously. Multicore processors are example of MIMD category.

General-purpose computers largely adopt the MIMD model for parallelism. With its widespread implementation, MIMD computers can be further classified into two categories according to two aspects of their memory organization: the virtual and the physical memory. Processors in MIMD machines could have physically shared memory, called multiprocessors, or physically distributed memory which are called multicomputers. From a virtual view of the memory, MIMD computers could use shared address space or distributed address space. The two views of the memory (physical and virtual) need not be the same; a system with shared virtual address space can run on top of physically distributed memory.

3.2.1 Distributed Memory Architectures

A distributed memory system consists of a number of nodes, each node contains a processing element, local memory, and possibly I/O elements. Nodes are connected via an interconnection network that communicates data between nodes. Data stored in local memory is private to its processor. When a processor needs data from other nodes, it typically exchange send/receive messages with the target node. Therefore, message passing is the preferred parallel programming model for distributed memory systems [Rauber and Rünger, 2010b].

3.2.2 Shared Memory Architectures

A shared memory machine consists of a number of processing elements, a global shared memory, and an interconnection network to connect processors with the global memory. The global memory in modern machines is implemented as a set of multiple memory modules. Data communication between processors is performed by writing or reading from shared variables. Write operations to shared variables must not be concurrent; otherwise a race condition would occur with an unpredictable result.

The shared memory model can form two different architectures with reference to the memory access latency: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) architectures. In UMA architecture, processors are connected to the shared memory via a central bus, *Front-Side Bus*. The distance between processors and the shared memory is uniform; therefore processors have equal access latency to the memory. This central bus provides constant bandwidth to all processors; thus it increases access collisions and causes additional access latency. In addition, there is no private memory for processors, but they use a cache hierarchy to accelerate access to data. Hierarchical organization of processors, cores, and caches imposes communication overhead between processing elements based on the distance between each other [Cruz et al., 2010]. For instance, the AMD Bulldozer micro-architecture incorporates a shared L2 cache between every two cores. A data placement policy should consider such a design to improve cache efficiency.

A Symmetric Multiprocessor (SMP) is an implementation of UMA shared memory architecture, where multiple processing elements "cores" are integrated in a single die [Gepner and Kowalik, 2006]. Each core is considered as a processor and has its own resources. In addition, each core has its own cache subsystem and may share part of the cache hierarchy with other cores in the same die. SMPs usually employ a small number of processors because adding more processors to the SMP chip would increase memory access collisions on the central bus. Consequently, scalability of SMP processors is limited [Esmaeilzadeh et al., 2012]. The maximum number of processors in a bus-based SMPs is between 32 and 64 [Rauber and Rünger, 2010a]. Figure 3.2 depicts a standard multicore UMA architecture.

For more scalable SMP processors, a processor can integrate a number of cores in a single chip and distribute the memory among the cores. The memory address space is shared between cores, however, the memory access latency is non-uniform. A core may exhibit long access latency time to access remote memory. To minimize remote memory access overhead, the cores may use a cache subsystem. Cores must ensure that a memory address contains the most recently updated value by using a suitable cache coherency protocol. This kind of architecture is called cache coherent NUMA (ccNUMA). Figure 3.3 presents an example of multi-hop NUMA architecture.

The trend for modern chips is toward less memory per core [Vajda, 2011]. When multi-

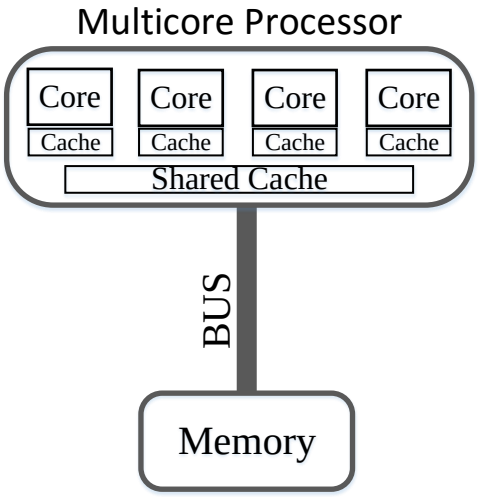


Figure 3.2: A standard multicore UMA architecture diagram.

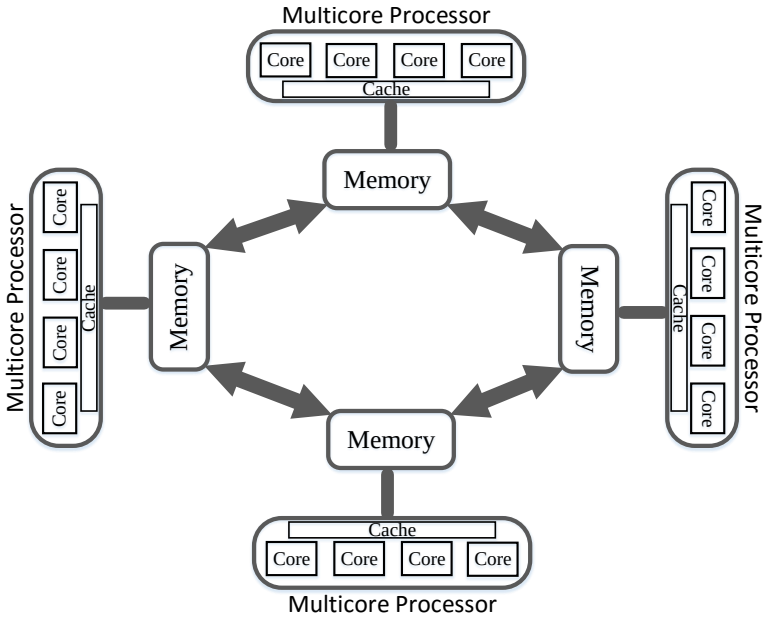


Figure 3.3: A standard multi-hop multicore NUMA architecture diagram.

threaded applications consume per-core memory, the rate and time for freeing memory is likely to be high. This research attempts to improve data locality for garbage collection to reduce collection time. Therefore, improvement to garbage collection would have a greater impact on modern processors.

The next section discusses the NUMA architecture in more detail, since it will be the focus of this research.

3.3 Non-Uniform Memory Access Architecture

Modern multicore processors implement a distributed shared memory model that takes the form of multi-node multi-socket system. Every node consists of a number of cores attached to "*local*" memory banks. Usually, a processor socket represents a single node. An AMD Opteron processor that we use in this research contains two nodes. Sockets are connected by a network of links. The design of the processor's interconnection is a processor manufacturer propriety. For instance, AMD uses HyperTransport™ technology and Intel® uses QuickPath interconnection technology to connect nodes. In NUMA architectures, cores in each node have uniform access latency time to the local memory. When a core accesses non-local *remote* memory addresses, it incurs additional access latency due to the off-chip interconnections delay between nodes.

NUMA architectures create opportunities to scale up the number of cores and the memory bandwidth. This research uses a NUMA AMD system, which consists of four sockets and eight nodes. Each node contains eight cores and 64GB memory. It implements the "Piledriver" micro-architecture, which has 2MB L2 cache shared by every two cores and 6MB L3 cache shared by all eight cores in a node. This hardware specification is taken from Linux "/proc" files. Figure 3.4 depicts the AMD Opteron NUMA multi-core system.

AMD Opteron processors manage memory access transactions through a dedicated unit called the NorthBridge (NB) [Conway and Hughes, 2007]. Figure 3.5 depicts a NB micro-architecture design. This unit is responsible for routing the memory transactions originated from cores and interconnect links to access core, cache, DRAM, or interconnect links. When a memory access request misses the L3 cache, which is the Last Level Cache (LLC), the request is sent to the NB unit. Since memory banks are distributed, the physical address is mapped to the *normalized address*. Memory is accessed by the normalized address only and the NB unit is responsible for translating the physical to normalized addresses.

The NB unit includes various components:

1. System Request Interface (SRI): holds a table of physical to normalized address mapping.

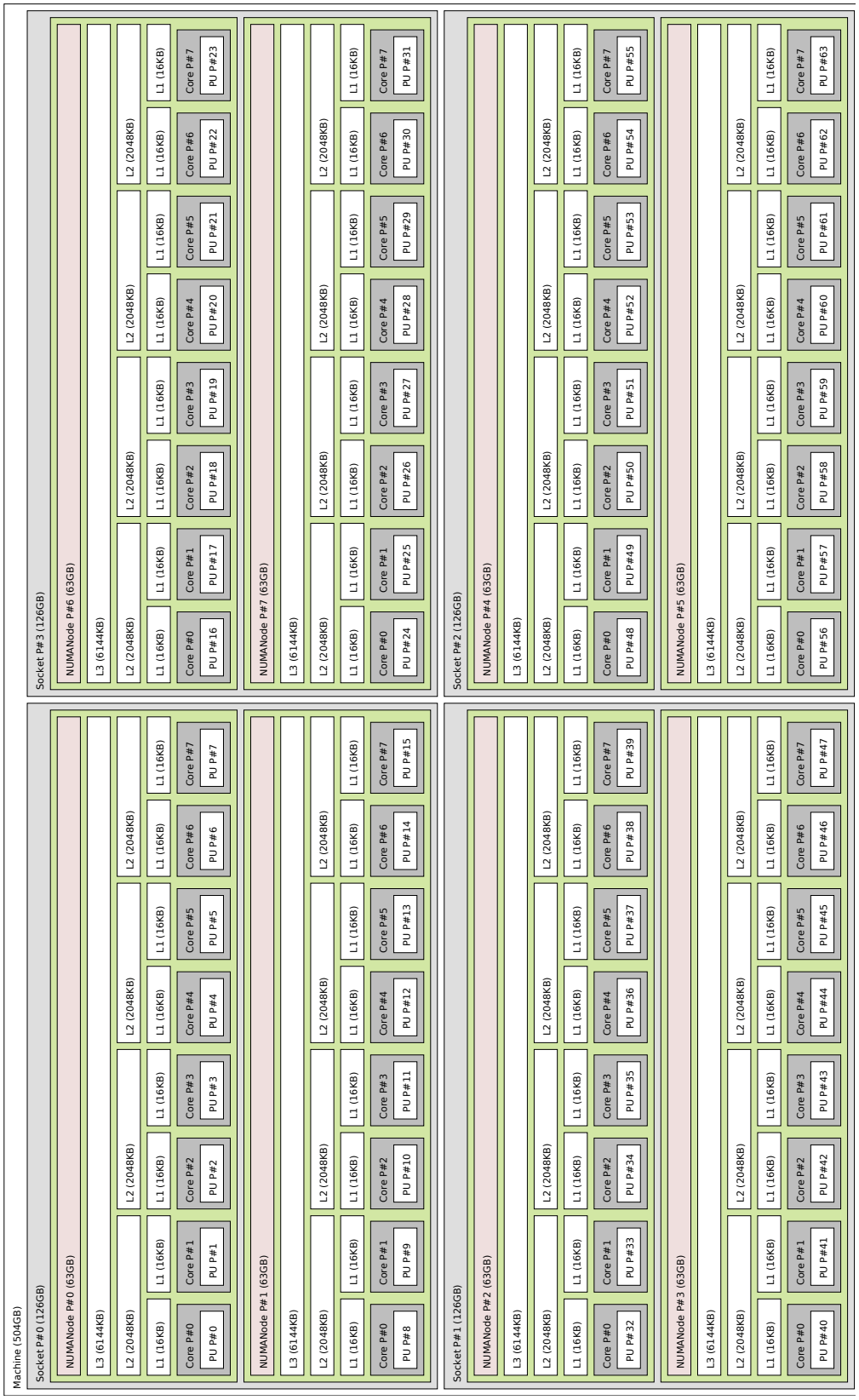


Figure 3.4: AMD Opteron 6366 NUMA architecture topology. This diagram is generated by lstopo tool [Broquedis et al., 2010].

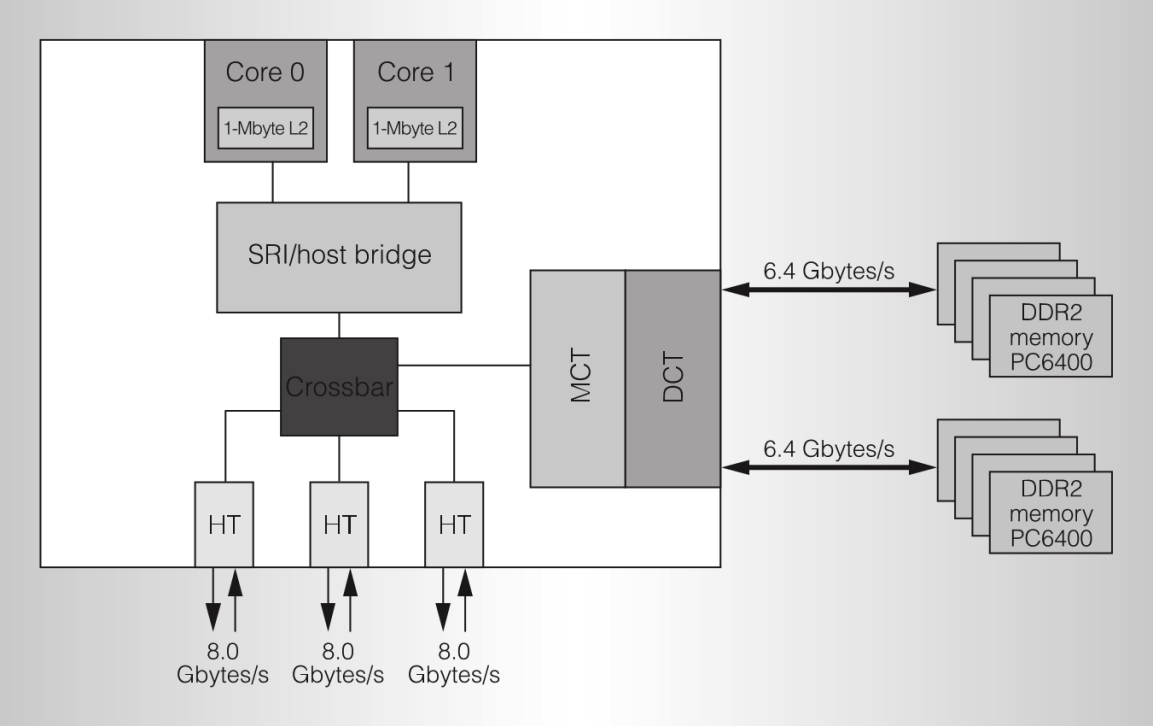


Figure 3.5: AMD NorthBridge micro-architecture. Source [Conway and Hughes, 2007]

2. Memory Controller (MCT): is responsible for managing the data flow from and to the main memory and contains the logic to read and write to DRAM.

3. DRAM Controller. (DCT) controls memory transactions to memory banks, for example DRAM refresh operations.

4. HyperTransport link ports: The NB unit has three ports to connect to other nodes.

A typical memory access journey is as follows: suppose a core is executing an instruction that needs to access data in the memory. If data is not in the cache hierarchy, the memory access request is sent to the SRI component. The SRI decodes the physical memory address to generate the normalized address. In addition, the SRI checks whether the memory address belongs to its local memory ranges. If the memory access is to local memory, then the SRI sends the memory request to the on-chip memory controller and waits for the memory transaction to complete. Otherwise, it looks up the routing table and forwards the memory access request to the appropriate HyperTransport link port and then awaits for the end of the memory transaction.

The overhead of accessing remote memory location begins at the stage of looking up the routing table to find the destination node. The remote access latency counts the round trip of a complete memory transaction. In addition, the off-chip interconnection network in our system involves multiple access hops. A memory transaction may get routed twice to reach

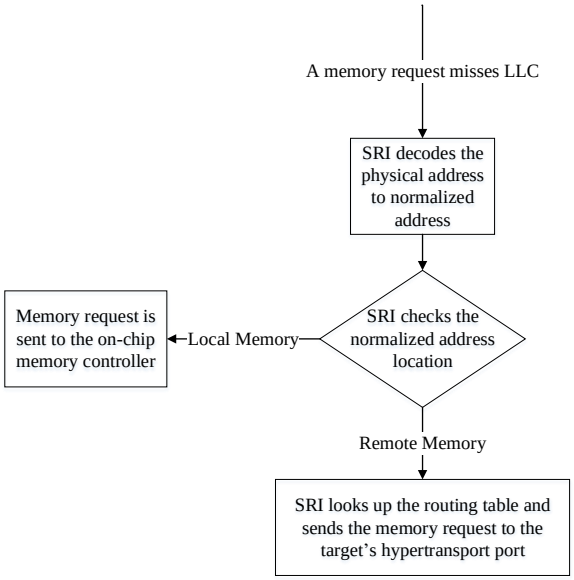


Figure 3.6: Memory Request

the final memory destination. This dissertation attempts to minimize this remote memory access overhead by managing garbage collection threads to access local memory only.

3.4 NUMA Memory Allocation Policies

As discussed in Section 3.3, processors in a multi-hop NUMA system may access distant memory locations and incur additional access latency time. NUMA-aware operating systems provide various memory allocation policies. The default memory policy in Linux is called *First-Touch* policy. Setting a memory policy for a process or a range of memory addresses does not take effect until the page is hit by a memory transaction. Untouched memory pages of a process are not allocated. If the untouched page is requested for access, the processor issues a page fault. The Linux kernel handles this page fault and allocates the page according to the configured memory policy, i.e. assigns the page to a NUMA node. The instruction that requested an access to that page is restarted and is able to access the memory.

Two criteria influence memory allocation policies: *locality* and *balance*. Local node memory allocation policy is the most common policy. The local node policy attempts to grant local cores fast access to memory by allocating memory from the local node and avoiding off-chip interconnection delay.

However, a multi-threaded program may allocate and access large amounts of memory in its sequential code segment. Alternatively, it may be biased towards a small number of cores enabling them to allocate much of the program's memory needs from a few nodes. This

imbalanced memory consumption may saturate the local node's memory capacity and bandwidth while other nodes still have enough resources. As a result, operating systems introduce *interleaved* memory policy, which trades off the locality for memory balance. Interleaved memory policy allocates memory from the system's nodes in a round robin order.

Linux, since kernel 2.6, uses the interleaved policy as the default policy on boot-up for the kernel processes. Since kernel structures are shared among running processes, it is advantageous to distribute the kernel structures across all nodes. The interleaved policy avoids excessive load on a single node when processes access kernel data structures. The default policy changes to local node when the first userspace process is started [Lameter, 2013].

Processes inherit the default policy, which is local node, when they start running. The Linux scheduler prefers to keep the process running on the same node to benefit from cache locality. In particular, the scheduler leaves the process to run on cores that share L2 cache, then cores that shares L3 cache of the last run. At last and for load balancing needs, the scheduler will move the process to another node.

Operating systems provide tools to control processes with a specific NUMA execution environment. The main Linux tool is `numactl`, which can display the system's NUMA configuration and set a user-specified NUMA scheduling or memory allocation policy. When a policy is set for a process, all its children inherit the same policy setting. The Linux `man numactl` command displays the tool's usage and examples.

3.5 Virtual to Physical Memory Page Mapping

Operating systems execute a computer program in an autonomous entity called a process, which is a dynamic instance of a program. Modern computer machines provide physical resources with high capacity, for example manycore processors, large memory space, and video accelerators. Therefore, many processes can run simultaneously by sharing the system resources. One of these resources is the memory, and operating systems incorporate many techniques to organize the memory usage.

Since every process is an autonomous entity, a process creates its own linear and contiguous virtual address space to allocate program code and data. The process treats its virtual memory as if it is the sole owner of the physical memory. The main advantage of using virtual memory is to enable many processes to run at the same time without any interference between the processes.

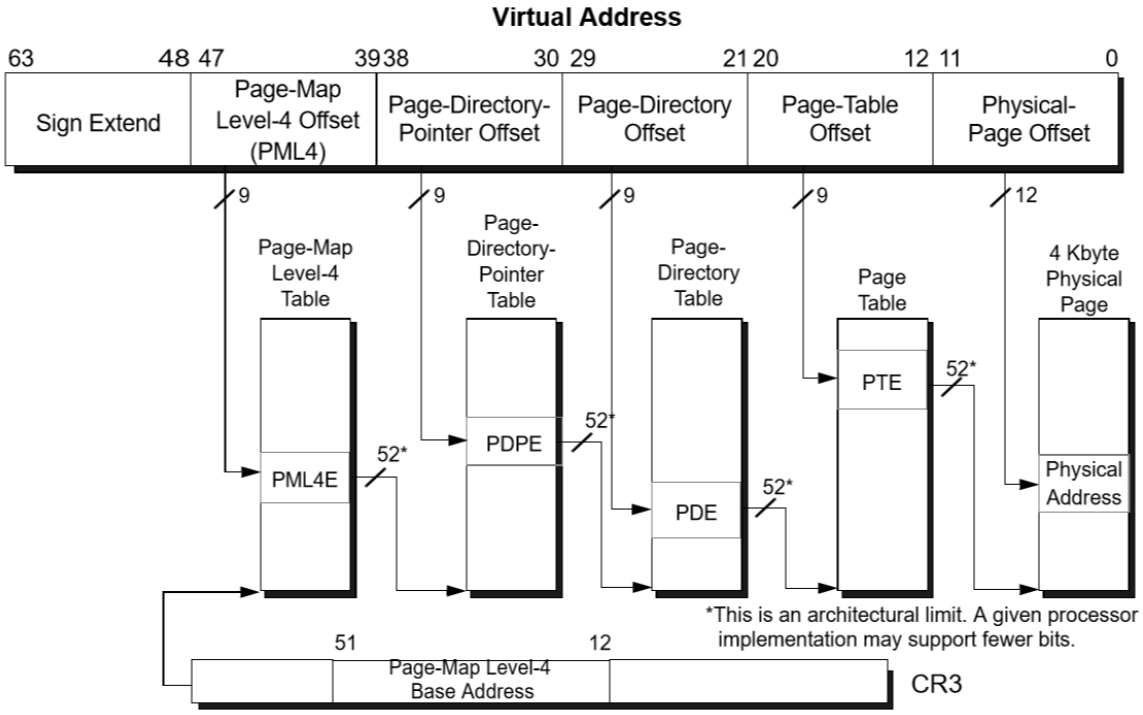


Figure 3.7: Hierarchical Page table for 4KB page size on 64-bit x86 Linux. Source [AMD, 2015]

3.5.1 Memory Pages

The physical memory is divided into chunks of memory called *frames*. Operating systems set different sizes for memory frames. For instance, Linux sets the default physical memory frame size to 4KB. Bigger frame sizes are available and the system developers provide various configurations to suit different needs. Linux also provides 2MB and 1GB memory frames.

Processes allocate code and data in the virtual memory *pages*. When a process attempts to access a memory location, the operating system needs to translate the virtual memory address to a physical memory address. The virtual address is divided into two segments: a virtual memory page and an offset address. The translation operation is done with the help of a hardware cache component called Translation Lookaside Buffer (TLB), which stores the recent memory mapping entries. If the virtual memory page is not cached in the TLB, the operating system issues a page fault signal to retrieve the translation entry from the page table (*page walk*), which contains virtual to physical mapping entries. The operating system walks through the page table and searches for the mapping entry of the requested virtual memory page. If the page is touched for the first time, the operating system allocates a physical page and updates the page table. Once it retrieves the mapping entry, it updates the TLB cache and restarts the memory transaction. This mechanism is called *on-demand paging* to manage the physical memory efficiently between the running processes.

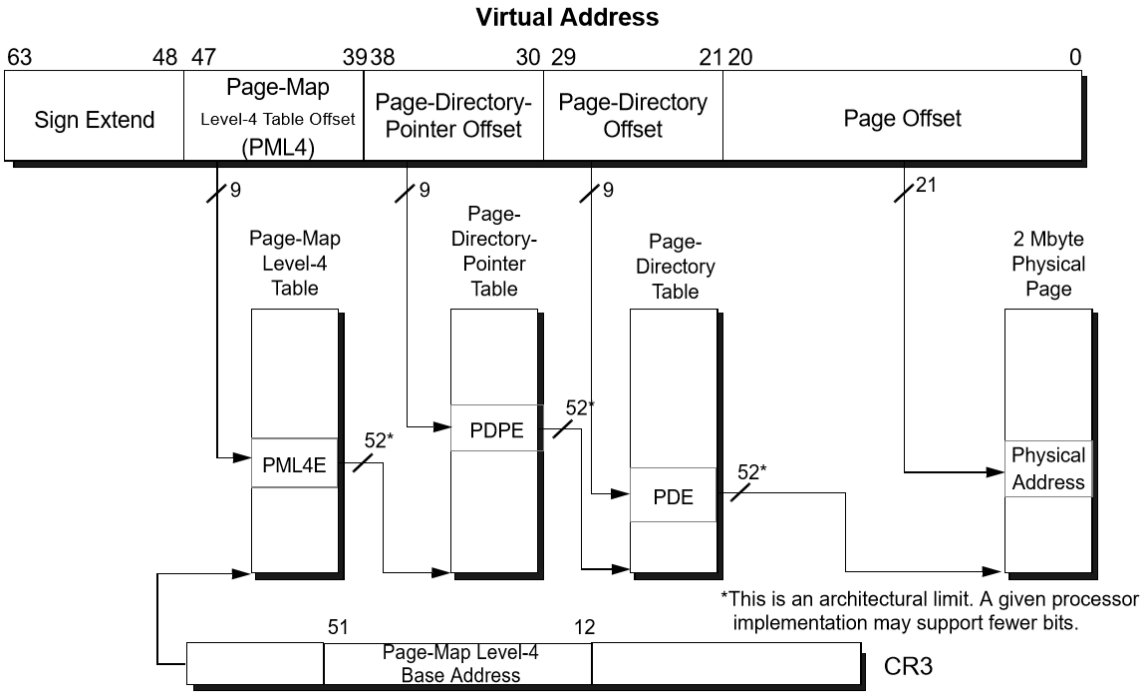


Figure 3.8: Hierarchical Page table for 2MB page size. Source [AMD, 2015]

For 64-bit processor architecture, the address space is huge. AMD Opteron uses 48 bits for physical address space and 48 bits for virtual memory address space. Walking the page table of such a huge memory spaces incur space and time overhead. For example, for 48 bit virtual memory page address and 4KB page size, we get 64GB (36 bits) of page table entries. Therefore, operating systems incorporate a *hierarchical page table*, which implements multi-level paging tables to manage the page translation entries in a more efficient way. AMD Opteron processors implement four paging levels as described in Figure 3.7. The virtual address contains the address offset (12 bits) and other paging level fields (36 bits). Thus, walking the page table with a hierarchical page table requires less space when compared to the flat paging mechanism.

Memory-intensive and big data applications require large memory capacity that the system provides. These applications allocate large data structures, which cross the memory page boundaries and use many memory pages. Although the virtual memory space for such data structures could be contiguous, the physical memory space is unlikely to be contiguous. For NUMA systems, these memory pages could be mapped to different NUMA nodes. In addition, every memory access requires a translation operation and if the page entry is not cached in the TLB, accessing a data structure would require many page table walking operations. This additional overhead may degrade the application performance.

Operating systems provide a solution to minimize memory page mapping and page table walking. Instead of dividing the physical memory into frames of size 4KB, the frame size

Memory Allocation policy	No. of Pages
Interleaved	290 (4KB) 49 (2MB)

Table 3.2: First 290 virtual pages were mapped to memory nodes in a round robin order. Then, transparent huge pages are used to map every 512 virtual page to a memory node.

could be of size 2MB or 1GB for large memory space. In the case of 2MB memory page size, the virtual address contains the memory address offset (21 bits) and other paging level fields (27 bits). Figure 3.8 depicts the virtual address fields.

Applications may use the *hugetlbfs* file system to allocate data in huge pages. Although huge pages improve locality by co-locating data in the same page, they may impose internal fragmentation when part of the page is not being fully utilized. Furthermore, operating systems provide huge pages as an optional configuration, and the system administrator needs to mount the file system to use huge pages.

Alternatively, Linux implements Transparent Huge Page "*THP*" mechanism to support automatic promotion and demotion of frame sizes. If THP is enabled, the operating system maps the virtual memory page to a huge frame without user intervention. In contrast to *hugetlbfs*, THP does not need to reserve space for huge pages. In addition, applications require no modification in the code to take advantage of THP.

To examine this mechanism on our target platform, a micro-benchmark is written to allocate a large object and outputs the memory mapping results. The memory policy was set to interleaved and the program allocates a 100MB object. At each 4KB page boundary, the code retrieves the NUMA node of each memory page.

The results show that the operating system mapped the first 290 memory pages to the NUMA nodes in a round robin order. Subsequent virtual memory pages were mapped to physical frames using THP mechanism. Table 3.2 illustrates the results. Accordingly, the operating system uses the remaining 4KB page entries in the current page directory then switches to huge page entries for subsequent virtual memory pages. The remaining 4KB pages in a page directory varies and each process may get different number of 4KB pages.

3.6 Java Virtual Machine and Garbage Collection

The Java Virtual Machine (JVM) is an abstract computing machine defined by the Java specifications "JAVA SE" [Gosling and Buckley, 2015]. Hotspot is an implementation of the JVM released by Oracle Corporation. There are many JVM implementations, for example IBM J9, Jikes RVM, and Android Runtime (ART). OpenJDK is an open source project by Oracle to enable the Java community to contribute to Java standards and implementations.

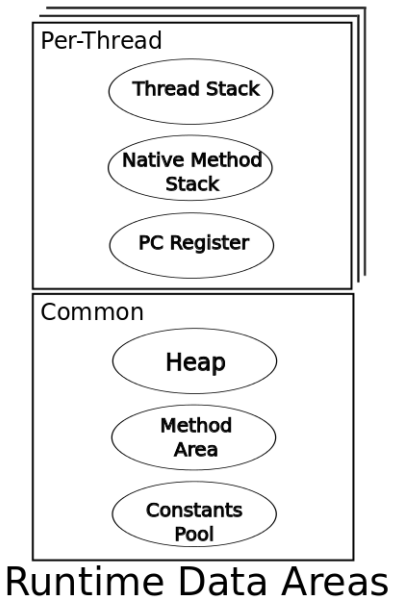


Figure 3.9: JVM data areas

OpenJDK Hotspot JVM implements many components, for example bytecode interpreter and Just In Time (JIT) compilers and three garbage collectors. This dissertation extends and modifies Hotspot JVM of OpenJDK version 8 [OpenJDK, 2015].

The JVM defines various runtime data areas. Each JVM thread has a stack which lives for the duration of the thread lifetime. A thread's stack holds the local variables and partial results. All JVM threads share a runtime heap, where dynamically allocated data is stored. They also share a method area, which stores per-class structures and code. The constant pool area stores per-class or per-interface constants. Native methods that are written in different languages are stored in native method stack area. Figure 3.9 depicts the JVM data areas

The runtime heap area implementation in the OpenJDK Hotspot JVM is a generational memory, where the heap is divided into age-based generations. General garbage collection concept are presented in Section 2.1. The Hotspot JVM implements three different garbage collection policies: Parallel Scavenge, Concurrent Mark-Sweep, and Garbage first. These garbage collection policies use parallel threads to collect the heap. When the application threads and the garbage collection threads execute in the same time, the garbage collection is called "concurrent". In this dissertation, we focus our optimizations on the **Parallel Scavenge** (PS) policy. PS collector is a Stop-The-World collector, where the application threads need to pause before the garbage collector threads commence collection.

PS splits the heap into two generations [SunMicroSystems, 2006]. First, the young generation accommodates immature (young) objects and is exposed to *minor* collections in order to preserve the live objects that are still needed by the application and reclaim the memory occupied by garbage. It consists of two spaces: the Eden space, which accommodates newly

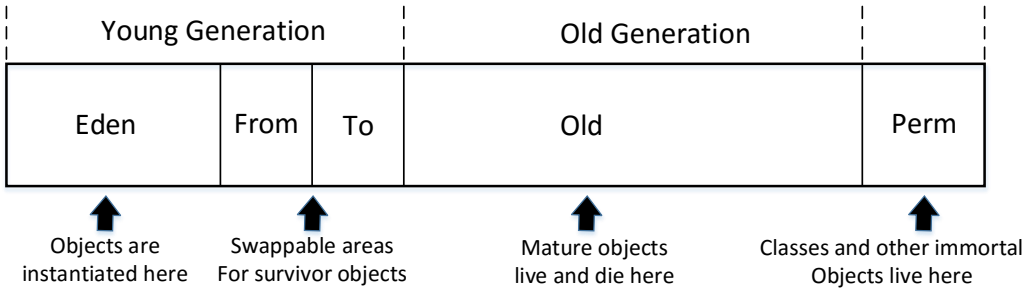


Figure 3.10: A schematic view of the heap spaces: Eden and survivor spaces (the young generation) and the old generation

allocated objects and the semi-space survivor space to hold survivor objects of the minor collections. Second, the old generation retains mature objects that live for long time. Jones et al. [2011] explain garbage collection notions and concepts in more detail.

The PS policy implements a copying collector that evacuates survivor objects from the Eden space to the survivor space or from the survivor space to the old generation. For the old generation, the PS implements a Mark-Compact collector. Live objects are marked first then compacted to reclaim the garbage memory. Figure 3.10 illustrates the PS heap structure and garbage collection policies for each generation.

Every mutator thread allocates a Thread Local Allocation Buffer (TLAB), a large, private buffer in the Eden space. TLABs improve memory allocation performance because synchronization on the global heap lock is minimal when large buffers are allocated. Moreover, mutators co-allocate objects contiguously in the TLAB and this mechanism improves spatial locality. The existing minimum buffer size is set to 8KB and the memory manager expands and shrinks TLABs according to the JVM ergonomics.

Similarly, garbage collector threads use large buffers to place survivor objects. Every thread allocates a Promotion Local Allocation Buffer (PLAB) in the survivors space and in the old generation. The default PLAB size in the young generation is set to 16KB, whereas it is 4KB in the old generation.

The next section discusses the copying collector in detail.

3.6.1 The Copying Collector

Short lived objects reside in the young generation, thus, many dead object are likely to fill the young generation. When an allocation request fails due to the lack of free memory, the memory manager triggers a minor collection operation to collect the Eden space and evacuates surviving objects to the survivor space or to the old generation. The space left in

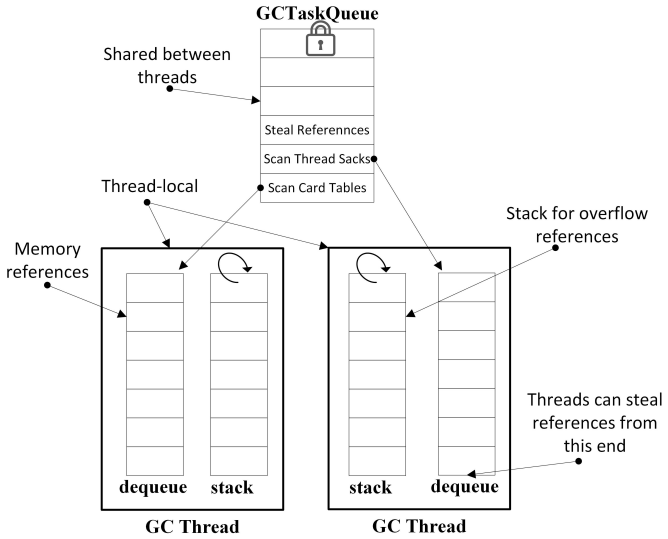


Figure 3.11: A diagram of the GCTask and thread-local data structure.

the Eden space would be a large contiguous memory for future allocations.

Data Structures

The copying collector uses various data structures to manage the collection operations. First, `GCTaskQueue` is a *shared deque*, which uses the Arora queue design [Arora et al., 2001] Arora queue implements single producer/multiple consumers paradigm. From one end, only one thread can enqueue, whereas, multiple threads can dequeue from the other end using atomic operations. This queue implements the Arora design [Arora et al., 2001].

At one end, a thread called the Virtual Machine (VM) thread populates the queue with `GCTasks` during the sequential phase. GCTasks are tasks with different functionalities and the three main types are: root scanning, work stealing and finalizing tasks; these tasks are explained later.

When the parallel phase begins execution, the collector threads may push GCTasks of other types to manage the parallelism. Therefore, this queue's end must be wrapped with a lock to synchronize thread access. The other queue's end is lock-free and the collector threads use atomic operations to pop GCTasks.

Second, every collector thread creates a private deque to iterate over root closures. The Arora queue enables the queue to push and pop from one end. The other end is left for starving threads that run out of work to steal references and balance the load across participating threads. To avoid space overflow, the Hotspot JVM creates a stack accessed by the owner thread only to push references when the Arora queue is full.

Figure 3.11 depicts a schematic view of GCTasks and thread-local queues.

Third, the VM thread pushes different types of GCTasks into the GCTaskQueue. There are three main GC task types that manage the parallel collector thread execution.

Root Scanning Task: A root scanning task directs a garbage collector thread to a memory area, where potential root references can be found. For example, root references may reside in TLABs, Java Native Interface (**JNI**) handles, or card tables; therefore, the VM thread creates many root scanning tasks and pushes them into the GCTaskQueue. The card table refers to a table that records inter-generational references. References from the old generation to the young generation is recoded in the card table.

Work Stealing Task: The collector threads process references by the means of closures in which a root reference and its sub-graph are copied to other heap spaces. However, the depth of root closures is not equal; deep closures require additional time for processing. Consequently, the VM thread creates work stealing tasks to maintain the load balance across the collector threads. The number of work stealing tasks is equal to the number of the collector threads.

Finalizing Task: At the top of the queue, a unique task is used by the termination protocol to synchronize the collector threads and hand over the control to the VM thread.

Algorithms

The garbage collection begins with a sequential code phase executed by the VM thread for preparing, bookkeeping and managing the collection operations. The collector threads park at the GCTaskQueue monitor, waiting for GCTasks to process. As described earlier, the GCTaskQueue is prepared in this phase and the VM thread hands in the control to the parallel phase and sleeps.

The parallel phase starts by waking up the collector threads, notifying them that the GCTaskQueue has GCTasks ready for processing. The collector threads compete on the queue lock and the successful thread pops a GCTask, which should be a root scanning task. Other threads spin on the lock, trying to lock the queue and pop a GCTask. Once root scanning tasks have been processed, every collector thread pops a work stealing task and attempts to get work from busy workers. The remainder of this section focusses on two parallel techniques: work processing and work stealing.

Work processing begins by popping a root scanning task from the GCTaskQueue. This task directs the collector thread to a memory area and scans resident objects to find ones that have a reference to the heap. These objects are considered as roots and are copied to the thread's

local Arora queue. When the collector thread completes scanning the memory area, the local queue begins to be ready for processing the root references and their attached sub-graphs.

The collector thread pops a reference, copies it to the target heap space, and scans it if it has a reference. References that are discovered while processing the root closure, are pushed into the thread's local Arora queue. The traversal order is depth-first, thus children and parents are co-located together. If a collector thread comes across a referenced object that already has been copied, it skips that object and pops another reference. Once all references in the local queue have been processed, the collector thread pops another root scanning task, if any, or a work stealing task.

A work stealing task enables the collector thread that has finished root scanning tasks to peek on other threads' queues. The existing stealing algorithm selects two queues randomly and compares their sizes. The larger queue size is chosen and the stealing thread atomically pops a reference from the end. The stolen reference is processed in the same way described for root scanning task. In the case when a stealing thread fails to obtain a reference, it repeats the process many times until it reaches a threshold. Hotspot sets the threshold equal to double the number of participating threads.

While the collector threads work on root scanning and work stealing tasks, one thread pops the finalizing task and gets promoted as a leader to terminate the parallel phase. Every collector thread increases a global counter atomically to indicate that reference processing work has been completed. The finalizer thread checks the counter and if all threads have finished their work, it wakes up the VM thread and hands over control to it. All garbage collector threads return and park at the GCTaskQueue monitor. The sequential code executes the remaining collection operations and resumes application execution.

3.6.2 The Mark-Compact Collector

The old generation contains long-living objects and spans a large memory space compared to the young generation. It is collected by a mark-compact garbage collector which eliminates fragmentation and compact live object towards one side of the heap. The mark-compact collection algorithm consists of four phases:

1. **The Marking Phase**: The heap is split into regions to divide the work across the compactor threads. At each region, a marking thread marks all live objects and calculates their total size.

2. **Summary Phase**: A single thread sequentially calculates the object's size in each region and sets each object's destination. In addition, it calculates the dense prefix, which is an area that holds old-enough objects that are likely to be eternal; hence, cannot be moved.

3. **Compaction Phase**: At this stage, live objects are moved to their new location and reference addresses are updated. Updating the address of objects that cross two or more regions are deferred to the last phase. Compaction is performed on a region basis; regions that are ready to be filled are pushed into a shared queue and the compactor threads atomically pop a region and compact objects into it.

4. **Clean up Phase**: It updates the references of deferred objects and re-initializes the variables.

Data Structures

The mark phase uses the same data structures, i.e GCTasks, GCTaskQueue, and thread local Arora queues, as in the copying collector. In addition, Hotspot uses a Mark Bitmap data structure to mark the objects live using atomic operations. The compaction collector assigns regions to the parallel collector threads by adding ready to fill regions in an Arora queue.

Algorithms

The marking phase performs the same operations as the copying collector except that instead of moving objects to their new location, it marks objects live. When a marking thread pops a reference from its local Arora queue, it sets the corresponding bit in the bitmap and calculates the object size. The finalizing task maintains the parallel termination protocol and hands over the control to the VM thread to execute the summary phase.

The summary phase prepares the heap regions to compact objects in the same region or in other regions. At least one region is pushed into the shared Arora queue to begin the parallel phase. The compaction collector attempts to create a dense prefix which contains mature-enough objects that are likely to live for the duration of the program execution. This is done gradually every time the major collection is triggered. Objects are moved and compacted to one end of the heap. Objects in the dense prefix are not moved since they are already compacted. However, references may get updated to reflect new object locations.

3.6.3 The Parallel Scavenge Optimizations for NUMA Machines

The Hotspot JVM extends the Parallel Scavenge garbage collector to take advantage of NUMA machines [Oracle, 2016b]. The NUMA-aware allocator can be turned on with the `-XX:+UseNUMA` flag. When it is enabled, the Hotspot JVM divides the Eden space of the young generation into several segments and each segment is mapped to a NUMA node. Mutator threads allocate objects in TLAB buffers and those buffers are placed in the mutator's local node. Objects instantiated by a thread are most likely to be accessed by the same thread.

Other heap spaces, the survivor spaces of the young generation and the old generation, have pages interleaved from all NUMA nodes. The hypothesis behind this design is that threads would have equal access latencies for mature objects. Accordingly, objects are distributed across the memory nodes.

At the time of splitting the space into segments, the memory manager touches the memory pages of each segment. This eager memory access to the segments is to ensure that segments are mapped to the appropriate NUMA nodes. The heap resizing policy remains active to comply with the application's allocation rate; therefore, segments may expand or shrink as needed.

The implementation code [Oracle, 2016a] includes additional sanity check methods for segment's mapping correctness. For instance, it might happen that "remote" pages are placed in a segment. This is due to the fact that the target NUMA node has a shortage in its memory. Therefore, the memory manager scans every memory segment after each collection cycle to free the segment from remote pages. The NUMA topology may change during the program execution or threads are context-switched to another NUMA node. Consequently, after each collection cycle, the memory manager re-initializes the Eden space and applies the new changes.

3.7 Conclusion

This chapter has revised the technical details for the hardware and software infrastructure relevant to my experiments. They include NUMA multicore processor architecture and operating system support. It also highlighted the state-of-the-art OpenJDK Java virtual machine, with particular emphasis on the memory management system. All the details presented in this chapter are well known features of commodity NUMA systems available off-the-shelf. The next chapter will review in detail the academic research on NUMA platforms and JVM garbage collection.

CHAPTER

4

EXPERIMENTAL SYSTEM INFRASTRUCTURE

Throughout this dissertation, a variety of Java application workloads are evaluated. These experiments ran on a system which is built and configured to exhibit NUMA effects. This chapter describes the infrastructure used in the experiments, from both hardware and software perspectives. The setup described in this chapter is the basis for all experiments, unless explicitly stated elsewhere in the text.

The outline of this chapter is as follows: Section 4.1 details the hardware and operating system configuration. Benchmarks and heap size configuration are described in Section 4.2. Summary of this chapter is presented in Section 4.3.

This chapter provides the following contributions:

1. It describes the multicore machine used for this dissertation's experiments.
2. It describes a number of real-world benchmarks along with other crafted and modified benchmarks that give an understanding of program behavior.

4.1 Hardware Setup

NUMA access latency overhead's problem emerges when a system comprises several NUMA nodes. In particular, the overhead manifests if NUMA architecture is hierarchical which involves multi-hop communications between NUMA nodes. Such a system incorporates a wide spectrum of data placement options and multiple access routes. Experiments in this dissertation are being run on a machine with multi-hop eight-node NUMA architecture.

Our machine encompasses four AMD Opteron 6366 processors. A detailed view of the machine's processors and NUMA connections is depicted in Figure 3.4, page 41. This processor's family implements the *Piledriver* architecture which contains two cores in a compute module. A core is clocked at 1.8 GHz and the two cores have shared access to L2 cache (2MB). Each processor consists of two dies, each with 4 compute modules, making the total number of cores sixteen and the overall system 64-core machine.

Every die in a processor integrates a memory controller. Therefore, a processor holds two NUMA nodes and the system contains eight NUMA nodes in total. In addition, a memory bank of size 64GB is attached to each NUMA node, and the overall system memory size is 512GB. The processors are connected via four HyperTransport 3.1 links with speed of 3.2 GHz and throughput of 6.40 GT/s per link.

The machine runs Linux 3.11.4 64-bit kernel. We use *interleaved* memory policy, which implies that memory pages of size 4 KB are mapped to each memory node in a round-robin order.

4.2 Benchmarks

This section presents the benchmarks that are used as workloads in our experiments. They include a variety of Java-based applications from three benchmark suites: DaCapo 9.12 [Blackburn et al., 2006, 2008], SPECjbb2005 [SPEC05], and SPECjbb2013 [SPEC13]. These selected benchmarks are established benchmarks and they can enable us to evaluate the effect of NUMA overhead because of their large memory footprint. Furthermore, they are widely used and accepted in the field of memory management. There are other data intensive benchmarks which are more recent and could represent future parallel and high performance benchmarks, for example, graph500 benchmarks [Murphy et al., 2010] and Lonestar benchmark suite [Kulkarni et al., 2009]. They include real-world applications that exhibit irregular behavior and span various application domain such as meshing, clustering, and machine learning. However, these benchmarks target high performance systems and they are mainly written in C and C++, which are not garbage-collected runtime systems.

Benchmark	Heap Size (MB)	Input Size
SPECjbb2005	1000 (large)	N/A
GCBench	900 (large)	N/A
LiveJournal	100000 (large)	N/A
Avrora	16	Large
Batik	50	Large
Fop	40	Default
Jython	40	Large
Pmd	88	Large
Sunflow	40	Large
Xalan	40	Large
Eclipse	200	Large
H2	200	Large
Luindex	40	Default
Lusearch	40	Large
Tradebeans	200	Large
Tradesoap	200	Large

Table 4.1: Benchmarks and heap configuration

Our performance evaluation involves other benchmarks which are crafted or modified to widen the range of memory-intensive workloads. We modified Boehm's GCBench benchmark [Boehm, 2000] to support multi-threading and give more pressure to the memory [Singer, 2014]. Moreover, modern Java applications use graph-based back-end engines to store heap data as a graph. Therefore, we process a huge dataset which contains over 68 million nodes in a graph-based database using Neo4J and implement a complex query to find all possible paths between two randomly selected nodes.

The heap size is fixed close to minimum to invoke garbage collection many times. Table 4.1 summarizes the benchmarks and their configurations.

4.2.1 DaCapo Benchmark Suite

DaCapo benchmark suite is a tool for Java programming language and JVM benchmarking. It contains a set of open source, real-world applications with intensive memory workloads. There are two releases of DaCapo benchmark suite: 2006 and 2009. Benchmarks of DaCapo suite version 2006 with OpenJDK 7 are used for Chapter 5 and DaCapo suite version 2009 for the remaining chapters. The new version has more applications, which are multi-threaded and support concurrency for thousands of threads. Furthermore, the benchmark harness enables control over a range of input size for different evaluation needs. It also allows a configurable execution environment, for example users can select number of application threads, number of iterations to make the adaptive compiler to reach steady state

before measuring the execution time, and the ability for application chaining to test phase change. However, only seven benchmarks are compatible with JDK8, namely: avrora, h2, pmd, jython, lusearch, sunflow, and xalan.

Avrora

Avrora simulates a number of programs run on a grid of AVR microcontrollers. It is driven by a single external thread, but it is internally multithreaded with each simulated element using a thread (i.e. each node in a grid of simulated nodes is threaded). Avrora demonstrates a high volume of fine granularity interactions between simulator threads.

Pmd

Pmd analyzes a set of Java classes for a range of source code problems (18 code rules). It is driven by a single client thread; it is internally multithreaded using one worker thread per hardware thread. Pmd has irregular and complex object lifetime patterns.

Jython

Jython is a python interpreter written in Java which executes the pybench benchmark. It allocates around 1.2GB and has highly regular behavior. Jython is driven by a single thread, but internally it uses one thread per hardware thread. The tests performed are mostly single threaded.

lusearch

Lusearch uses the lucene library to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. Text size is about 10MB. Multithreaded, it is driven by one client thread per hardware thread, requiring little interaction between the threads. Each thread searches a large index for about 3500 distinct words. Lusearch allocates around 8GB memory [Yang et al., 2011].

Sunflow

Sunflow renders a set of images using ray tracing. It is multithreaded, driven by a client thread per hardware thread, each thread processing an available tile of work at a time and another thread created due to the use of the Java2D library.

Xalan

Xalan transforms an XML document which is the works of Shakespeare (5.3MB) and transforms the document into HTML. It is multithreaded, explicitly driven by the number of hardware threads available, each thread taking an element from a work queue. Xalan allocates around 60GB memory.

H2

H2 runs a JDBCbench-like in-memory benchmark, which executes a number of transactions against a model of a banking application. It replaces the hsqldb benchmark of the old DaCapo benchmark version.

H2 is multithreaded and it is driven by one client thread per hardware thread and internally has a server thread for each client thread as well as other support threads. The number of client threads for the default benchmark size is set to one per hardware thread.

4.2.2 SPECjbb (20XX)

SPECjbb2005 [SPEC05] evaluates the performance of server side Java by emulating a three-tier client/server system, which models an online shopping company. The benchmark exercises the implementations of the JVM (Java Virtual Machine), JIT (Just-In-Time) compiler, garbage collection, threads and some aspects of the operating system. It also measures the performance of CPUs, caches, memory hierarchy and the scalability of shared memory processors (SMPs). SPECjbb2005 provides an enhanced workload, implemented in a more object-oriented manner to reflect how real-world applications are designed and introduces new features such as XML processing and BigDecimal computations to make the benchmark a more realistic reflection of today's applications.

The application consists of several warehouses, each contains 25MB data. Each thread represents an active user requesting goods from the warehouse. Threads are mapped one-to-one to warehouses, in addition to the main method and the JVM functions threads. SPECjbb2005 measures throughput at a given number of warehouses. When the benchmark completes execution, it reports throughput at every measurement interval in terms of business operations per second (bops).

SPECjbb2013 [SPEC13] makes major changes over 2005 version. The new benchmark design involves multiple JVMs, which run concurrently, though it support a single JVM execution. The benchmark consists of three components: a controller which directs program's execution, one or more transaction injectors to issue transactions to the warehouses, and backends that contain business logic code to process requests from the transaction injector.

In this dissertation, we are going to use a single JVM configuration, which contains a single controller, a transaction injector, and a backend. The performance is measured by max-jOPS which is maximum number of successful jobs obtained. However, for both benchmark versions, we are more interested in the memory-intensive workload that frequently triggers the garbage collection rather than the throughput achievement.

4.2.3 GCBench

GCBench is an artificial benchmark which attempts to model allocation request's properties. It was written to mimic the phase structure that has been conjectured for a class of application programs for which garbage collection may represent a significant fraction of the execution time. The benchmark obtains time required to allocate and collect a balanced trees of various sizes.

Initially, GCBench warms up by allocating and then dropping a large binary tree. Allocated objects are of the same size and the total memory allocation is 32MB. After that, it allocates a large permanent tree and a permanent array of floating point numbers. Furthermore, it allocates considerable tree storage in seven phases, increasing the tree size in each phase but keeping the total storage allocation approximately the same for each phase. Each phase is divided into two sub-phases. The first sub-phase allocates trees top-down using side effects, while the second sub-phase allocates trees bottom-up without using side effects. GCBench produces a large amount of objects, increasing the garbage collection workload to a considerable extent.

In this thesis, GCBench is modified to test NUMA effect by enabling threads to allocate memory from different NUMA nodes. When the program starts, it sets up global data structures and prepare execution of concurrent worker threads. The program execution comes in three phases. First, every thread creates thread-local node objects. Second, at each tree depth, nodes allocated in NUMA node i point to nodes in NUMA node i+1. In this phase, pointers are shuffled between thread-local data structures. Third, threads exercises garbage collection with thread-local short and long lived data structures. In the end, the program reports the total execution time.

4.2.4 Neo4j / LiveJournal

Neo4j [Neo4J, 2015] is an open source, embedded, disk-based, fully transactional Java persistence engine that stores data structures in graphs instead of tables [Webber, 2012]. It is a scalable and high performance graph database that supports solving queries with complex relationships to store data in the nodes and relationships. Neo4j is written in Java and adopts

client/server programming paradigm. The graph nodes and relationships are loaded in the JVM heap.

Neo4j offers various robust mechanisms to store and retrieve individual nodes. In particular, it provides a fast traversal mechanism for a set of nodes based on their relationships, whereas conventional relational databases would require complex and sophisticated relationships. Many optimized functions are built into Neo4j for retrieving all, shortest, and user-defined path between two nodes.

To use Neo4j, we fill the graph with a data set from the LiveJournal social network. LiveJournal is a free online community with around 10 million members which allows members to maintain journals, individual and group blogs, and it also enables people to declare friendships. The data set consists of around 5 million nodes and 68 million edges [Leskovec and Krevl, 2014].

We implement a Java complex query program that embeds Neo4j 2.2.1 as a library and queries the database to find *all* possible paths between two randomly selected nodes. The program uses 64 threads to drive the workload and reserves 150GB heap size. The all-paths operation is repeated twice and the mean of N runs is reported.

4.3 Conclusion

In this dissertation, the hardware platform used for experimentation consists of eight NUMA nodes. It is selected to expose NUMA effect on application performance. Systems with few NUMA nodes might not be suitable for testing NUMA issues.

Benchmarking NUMA machines requires representative applications that have memory-intensive workload. The DaCapo benchmark suite obtains large memory footprint and provides flexible execution environment. However, DaCapo programs exhibit small heap sizes and modern NUMA machines employ massive memory capacity. Therefore, we include SPECjbb, GCBench, and Neo4J applications to widen our range of workloads. Recently, data-intensive benchmarks, for example Graph500, have emerged but they are maily written in C and C++ which are non garbage collected languages.

Part II

CONTRIBUTIONS

CHAPTER

5

A STUDY OF REFERENCE LOCALITY

Existing NUMA optimizations are applied at the granularity of individual objects, for example [Ogasawara, 2009, Zhou and Demsky, 2012, Gidra et al., 2015] . In these approaches, the garbage collector must track the NUMA node location of every live object at runtime. Therefore, the garbage collector incurs extra overhead which is proportional to the live object set size. The challenge is to reduce this overhead while maintaining NUMA awareness.

This chapter introduces a potential solution to minimize the NUMA awareness overhead. It presents a newly observed locality-rich characteristic exhibited by real-world application heaps. From the details of how the parallel tracing garbage collector traverses the reference graph, it is possible to describe the graph traversal mechanism in terms of a set of sub-graphs. Each sub-graph contains a root and some reachable objects. We call these sub-graphs **rooted sub-graphs**. Instead of inspecting every object, the *intrinsic locality* of rooted sub-graphs enables the garbage collector to retrieve NUMA node locations of a small object set, *the roots*. In this way, the garbage collector incurs less overhead as compared to previous approaches and this overhead is proportional to the root set size.

The main contributions of this chapter are as follows:

1. It introduces a previously unremarked locality feature in connected objects. Rooted sub-graphs exhibit natural locality that the garbage collector can leverage to improve its performance on NUMA architectures.

2. It studies heaps for a wide range of real-world applications to provide empirical evidence demonstrating that rooted sub-graphs exhibit high NUMA locality.

5.1 Introduction

When a program's heap space is exhausted, the runtime system invokes a garbage collector cycle to reclaim space occupied by the dead objects and move live objects to a different space. Live objects are those objects that are needed to continue program execution. The garbage collector treats live objects as a graph, *the reference graph*, where nodes denote objects and edges denote references between the objects.

Since the live object set is processed as a graph, object traversal and processing, i.e. copy or compact operations, can change the object location. As described in Section 2.5.2, graph traversal order plays an important role in object ordering and locality. The Hotspot JVM's Parallel Scavenge collector uses a depth-first algorithm to trace the reference graph. Depth-first algorithm co-locates children and parent objects in a single or continuous virtual memory pages. Although the garbage collector appears to improve object locality for an application, there are four issues that emerge in the context of NUMA architectures:

1. In NUMA architectures, the operating system could map virtual memory pages to different NUMA nodes; hence, objects that appear to be adjacent in the virtual memory pages, may be physically separated.

2. A garbage collector thread may traverse an object that is located in a remote NUMA node. There is neither enforcement nor preference for the garbage collection threads to process the local objects.

3. A garbage collector thread could move local objects (relative to the garbage collector thread) to a different NUMA node (relative to the application thread), which may cause application threads to incur remote access overhead, possibly for the rest of the program execution time. This counter-locality interest over the object location, clashes between the application and garbage collection threads.

4. The garbage collector processes the reference graph in units of sub-graphs. Each sub-graph represents a number of connected objects. However, parallel garbage collection involves work stealing as a parallel technique to speed up the processing. This technique could scatter connected objects across NUMA nodes if they are processed by remote garbage collector threads.

Existing NUMA-aware garbage collection techniques assist the garbage collector to preserve object locality interest for the application threads. In fact, the garbage collector may work

proactively and move an object to a NUMA node where it issues high volume of memory access requests to that object. Section 2.4 presents several techniques to move objects for the application benefit. Nonetheless, remote memory access burden may be incurred by the garbage collector threads.

A conflict of object locality interest between the application and the garbage collector threads can be mediated. Previous work, for example [Gidra et al., 2015, Zhou and Demsky, 2012], proposes that every garbage collector thread should process local objects only. This means that garbage collector must retrieve the NUMA node for every live object and communicate with another remote thread to process remote objects. Such a locality approach relies on message passing protocols, which usually incur additional overhead and may decrease the performance gains.

This chapter integrates both hardware and software technical knowledge to comprehend the reference graph locality. It studies the parallel garbage collection tracing algorithm and provides empirical evidence that modern NUMA machines tend to use huge memory pages (THP). Therefore, a high number of connected objects in a sub-graph reside in the same NUMA node. In addition, this chapter introduces a new locality heuristic, **the root location** that can reveal the NUMA node of a sequence of connected objects. Instead of inspecting every live object, this locality heuristic requires examining a subset of the live object set, *the root set*. Once a NUMA node of a root is retrieved, a large number of reachable objects from this root, reside in the same NUMA node as the root. Results from our benchmarks, which are described in Section 4.2, show that the percentage is 80% on average, with $\pm$ 7.9 error rate, see Section 5.6.2. The overhead incurred by object location inspection of the root set is low as compared to the default mechanism.

The outline of this chapter is as follows: Section 5.2 makes the case for rooted sub-graphs as an appropriate work unit for the garbage collector, which can show locality richness between their objects. In addition, it introduces a hypothesis that objects in a rooted sub-graph have a high likelihood to be in the same NUMA node as the root and this can be used as a heuristic to improve garbage collector locality. Section 5.3 and 5.4 describe our code modifications to evaluate the rooted sub-graph locality hypothesis. In Section 5.6, the root location heuristic is evaluated and empirically studied over a variety of benchmark applications. Section 5.7 contrasts our work with previous research and Section 5.8 summarizes this chapter.

5.2 Rooted Sub-Graphs

The garbage collector views live objects that require evacuation to other spaces in the heap as a reference graph. The reference graph is a type of directed graphs. Here are three definitions quoted from Rosen [1999]:

Definition 1 *"A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V".*

For the garbage collection, the set of vertices is the live object set and the edges are the references. The relation function in the reference graph is reachability.

Definition 2 *"Let G be a directed graph, and let v,$u \in V(G)$, we say that u is reachable from V if there exist a path from u to v in G".*

Initially, the only information known to the garbage collector prior to collection is a set of pre-defined memory regions, for example thread-local buffers and statics fields, where potential roots that have pointers into the heap can be found. Each garbage collector thread is assigned to one or more root areas to discover roots. The garbage collector, at this stage, is building the root set which contains the discovered roots. Once a garbage collection thread finishes the root *scanning*, it pushes its root references to a local queue.

A root object may have one or more descendant objects which form the transitive closure or the reachable set of a root.

Definition 3 *"Let R be a relation on a set A, the transitive closure R^n consists of the pairs (a, b) such that there is a path between a and b in R.*

Where R^n is a path of length n from a to b

Every object that belongs to a reachable set is called a *live object*. A shared live object can be reached from one or more roots. However, a live object is processed once, i.e. by only one garbage collection thread. At this stage, the garbage collector scans all discovered roots' transitive closures. More details of the Hotspot JVM's garbage collection implementation are presented in Section 3.6 47.

From the aforementioned description of reference graph processing, we can distinguish and cluster live objects into two groups: the root set, and the transitive closure of each root. Existing NUMA-aware garbage collection implementations treat live objects equally and inspect every live object to retrieve its NUMA node. In contrast, the reference graph can be viewed from a different angle. Each root from the root set is processed with its reachable set as a unit. We call this work unit a **rooted sub-graph**.

This research define rooted sub-graph as:

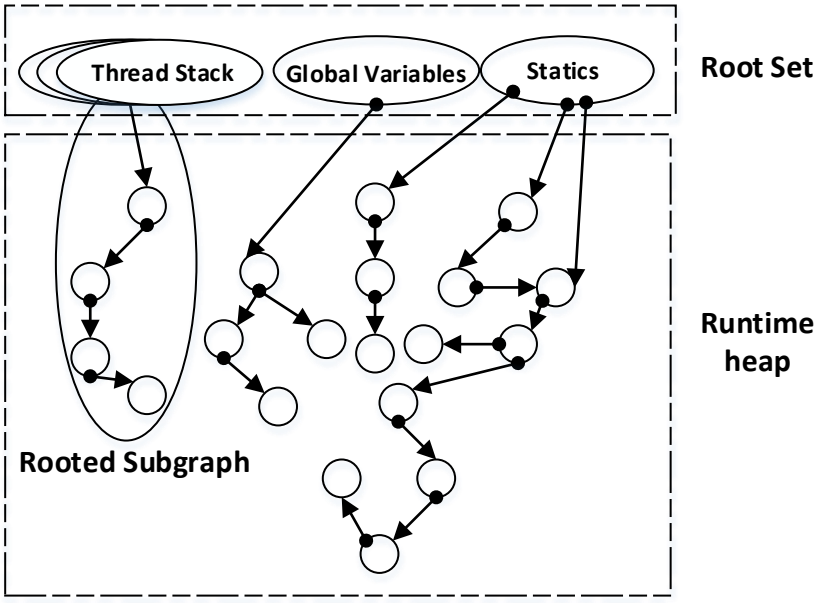


Figure 5.1: An example of a reference graph with different types of rooted sub-graphs

Definition 4 *A rooted sub-graph is a set of references, R, containing a root reference, r_{root}, with the condition that every reference $r \in R$ is reachable from r_{root}.*

A reference graph consists of a number of rooted sub-graphs. Figure 5.1 depicts an example of a reference graph with different types of rooted sub-graphs.

The set R is not necessarily maximal with respect to the reachable set formed by the transitive closure since some references are reachable from multiple root references. Intuitively, a rooted sub-graph denotes the set of references traced by a garbage collection thread from a root reference.

The garbage collection's graph traversal mechanism drives us to develop a hypothesis that a root reference with its reachable set has a *common* NUMA node location. The intuitions behind this hypothesis are two-fold:

1. The Hotspot JVM allocates buffers, i.e. TLABs, to every application thread to place data locally. The default TLAB size is less than 4KB, a virtual memory page size, which means that data allocated by an application thread is likely to be in the same memory page.

2. Even with a larger TLAB size, modern NUMA machines use transparent huge pages (THP) to reduce TLB buffer size, as described in Section 3.5. Therefore, a large virtual memory space, i.e. 2MB in our system, will be mapped to a single NUMA node.

The memory manager may allocate consecutive memory space for TLABs and these memory pages are likely to be in the same NUMA node.

Although these facts motivate in developing our hypothesis, there are other factors that affect the reference locality. The Hotspot JVM's parallel garbage collection implementation enables each root to be processed with its reachable set, up to a certain level. Thread-local Arora queues allow the root's owner thread, i.e. the thread that has pushed the root, to scan the transitive closure and push discovered references into the queue. However, if there is a garbage collector thread that has run out of work, it will try to steal work from *any* non-empty queue. In this case, part of the reachable set of a root will be processed by a different garbage collection thread, possibly mapped to a remote NUMA node.

To test the rooted sub-graph hypothesis, we need to obtain reference locality of the reachable set of each root for all rooted sub-graphs. The reference location information determines how wide/narrow is the reference distribution across NUMA nodes for each rooted sub-graph. Results can be summarized using mathematical and statistical means. In the next section, we will present our implementation to collect rooted sub-graph locality. Subsequent sections describe a locality metric that shows how a rooted sub-graph possesses a locality richness.

5.3 Implementation

The Parallel Scavenge garbage collection policy consists of a copying collector for the young generation and a mark-compact collector for the old generation. Rooted sub-graphs live initially in the young generation and then move to the old generation after several minor collections. Rooted sub-graphs may have references that cross the generations; however, the existing Hotspot implementation enables such references to be processed during minor collection. In this chapter, we analyse rooted sub-graphs that reside in the old generation only. The old generation is larger than the young generation, which may provide a variety of different NUMA-mapped memory pages. In addition, references in the old generation are amenable to several location changes due to the compaction algorithm, i.e. objects may live temporarily in multiple NUMA nodes throughout their life.

As a part of the major collection, the marking phase identifies live objects and calculates the live object set size. We have modified the marking phase to retrieve and report rooted sub-graph locality. Results are dumped into trace files, which contain objects' location distribution information. The reference graph during the marking phase is an artifact of previous copying and compaction operations. Therefore, different locality results are expected after each major collection cycle. This section will extend the marking phase implementation described in Section 3.6.2 and illustrate code modifications to generate the trace files.

In contrast to the copying collector, which scans an object and moves it to a different space, a marker thread scans an object and mark it as live without move operations. The Hotspot JVM uses a bitmap side table to store mark bits of heap objects. A marker thread that pops a reference from its local queue checks first whether this reference is for an already marked object. If that object is marked, the marker thread ignores the reference and terminates sub-graph marking, otherwise, it sets the corresponding bit in the bitmap table.

A rooted sub-graph's trace file is generated by adding an intermediate stage into the marking phase. In this stage, roots and their reachable set are classified according to NUMA node location. However; the existing implementation enables the collector threads to combine roots and their reachable object sets in the same data structure. To distinguish roots from their reachable sets, we make the runtime system to create as many Arora queues as NUMA nodes the underlying system has in order to store root references only. For instance, if the system consists of four NUMA nodes, the runtime system creates four NUMA queues. These root queues are created at the JVM initialization phase. When the JVM executes a major collection, marker threads scan root areas and before pushing roots to local queues, our code enables the marker threads to classify roots and push them into the corresponding NUMA queue. Once a marker thread completes root scanning tasks, it pops a root reference from its corresponding NUMA queue, retrieves its NUMA node, and records it in the trace file. After that, it scans the root's transitive closure (using a thread-local queue) and record the number of objects located in every NUMA node. The process is repeated until finishing all the roots. Stealing from non-empty queues is disabled to preserve the rooted sub-graph integrity. The generated trace file indicates the NUMA locality distribution for every rooted sub-graph.

5.4 Limitations

Tracing a rooted sub-graph that has a reference to a shared object would make a race between multiple marking threads. Only one marker thread wins a shared reference processing, which terminates other rooted sub-graphs at this reference. The winner marker thread carries on processing the remaining references in the sub-graph. In this scenario, our code may create different rooted sub-graphs for every run. Figure 5.2 illustrates an example of how marker threads create different sub-graph sequences. Rooted sub-graph for Rx and Ry intersect at node Z. All references that are reachable from Z will either be in the sub-graph for Rx or for Ry, which is a non-deterministic result. However, the set $Rx \cup Ry$ is deterministic which constitutes the live object set.

It is important to record reference locality for all possible sub-graphs even if shared sub-graphs exist. To tackle this problem, our experimental methodology ensures that test-bed

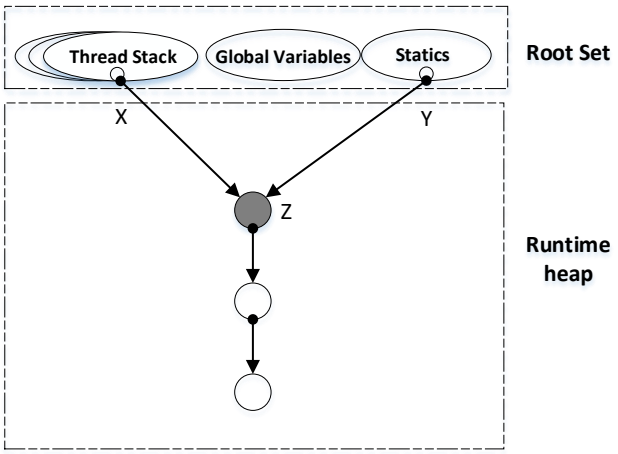


Figure 5.2: Rooted subgraphs for X and Y can be different depending on who first marks the node Z. The result is non-deterministic.

applications run for several times to minimize the effect of non-determinism. We emphasise the point that even part of the reachable set is set to one sub-graph, the other sub-graph still has a root reference and holds the definition of a rooted sub-graph. This scenario may continue to occur if the reference graph contains such connectivity, though we disable work stealing that truncates the sub-graph.

5.5 Experimental Setup

The modifications described in Section 5.3 are applied to the OpenJDK Hotspot JVM version jdk7u6. The Parallel Scavenge garbage collection policy, a stop-the-world collector (more details are described in Section 3.6.1), is used to generate rooted sub-graph locality trace files. The application and garbage collector thread count are set to 64 (the number of cores). Benchmarks run for five times to record rooted sub-graph locality measurements, and results are plotted with heat maps, as explained in Figure 5.3. The experiment reports the proportion of objects in each NUMA node instead of local / non-local nodes because the distance, and therefore the access latency, is different from one NUMA node to another. For example, to access NUMA node 1 from NUMA node 0, the memory access requires only one hop but from NUMA node 0 to NUMA node 3 it requires two hops. However, the effect of node distance is beyond the scope of this dissertation.

In addition to the heatmap presentation, we summarize quantitatively the NUMA locality of rooted sub-graphs with a scaler value. The proposed evaluation metric represents the locality richness in each rooted sub-graph. We retrieve the NUMA node of the root reference and also the NUMA node of each traced reference in the corresponding rooted sub-graph, then

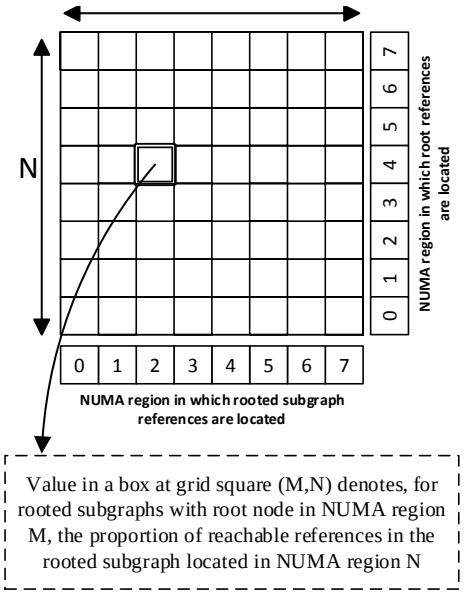


Figure 5.3: An explanatory example of heat map results

we calculate the percentage of NUMA-local objects in a rooted sub-graph.

For each workload, the locality of processed rooted sub-graphs is recorded in an n-by-n square matrix, where n represents the number of NUMA nodes. Matrix element a_{ij} represents the percentage of connected objects residing in node j that can be reached from a root in node i. We use the *Matrix Trace* property from Linear Algebra to calculate the NUMA locality of a workload. By definition, the trace of an n-by-n square matrix A is the sum of the elements on the leading diagonal, i.e.

$$tr(A) = a_{11} + a_{22} + ... + a_{nn} = \sum_{i=1}^{n} a_{ii}, \qquad 0 \leq tr(A) \leq n \times 100 \tag{5.1}$$

To illustrate how we can use matrix trace in our problem, let us take an example of the two extreme trace values. Our system has eight nodes; thus the trace value $tr(A) = 800$ represents perfect NUMA locality, where all references in the rooted sub-graphs reside in the same NUMA node as the root. For the other extreme, $tr(A) = 0$ means that all references in the rooted sub-graphs are in different NUMA node(s) from the root.

Due to the memory allocation policy and the workload behavior, some NUMA nodes might not be used at all. Therefore, we define the *Relative* NUMA Locality Trace metric such that:

$$loc(A) = \frac{tr(A)}{n \times 100}, \qquad 0 \leq loc(A) \leq 1 \tag{5.2}$$

where n is the number of nodes that contain roots.

For instance, a program p uses six nodes for object allocation and we calculate $tr(p) = 450$; thus $loc(p) = 0.75$ and we interpret the result as 75% of objects are allocated in the same node as the root.

5.6 Reference Locality Evaluation

In this section, we will evaluate the rooted sub-graph locality hypothesis on our chosen real-world benchmarks. These benchmarks include: multi-threaded GCBench micro-benchmark, various applications from DaCapo benchmark suite, and SPECjbb2005 benchmark. We have explored these workloads in Section 4.2. In addition, we will discuss the effect of garbage collection on rooted sub-graphs of SPECjbb2005 in more details.

5.6.1 Locality-distributed Rooted Sub-graph

Rooted sub-graph locality takes two forms. First, a *homogeneous sub-graph* consists of references, which point to objects located in one NUMA node. Second, a *scattered sub-graph* contains references to objects located in multiple NUMA nodes. This section analyzes rooted sub-graph locality of the multi-threaded GCBench micro-benchmark. For the purpose of this study, GCBench is modified to create locality-distributed sub-graphs. It instantiates 64 deep and long-living tree data structures. Objects at every tree depth are provided from an object pool in a remote memory node. The resulted sub-graphs contain references to multiple remote NUMA nodes.

Figure 5.4 depicts the results. The color scheme of the heat maps is shown on the right bottom corner of the graph. We can see two different locality-based groups of sub-graphs. First, scattered sub-graphs that have references to objects distributed across multiple memory nodes. In **GC cycle 1**, objects referenced by rooted sub-graphs touch almost all memory nodes. NUMA locality trace of this group is under 40% for nodes R_0, R_1, R_3, R_6, and R_7 , and around 80% for nodes R_4 and R_5. Second, a group of homogeneous sub-graphs in node R_2, which contain references to objects located in memory node 2 only.

To understand the locality of these groups, we examine two attributes: the size of the rooted sub-graph and the root type. The rooted sub-graph size is the total number of references a collector thread will process. The rooted sub-graph locality is the percentage of references located in the same root's NUMA node. Figure 5.5 depicts the correlation between the mean rooted sub-graph locality and the size of the rooted sub-graph. We can observe that rooted sub-graphs of size under 512 reference tend to have good locality; i.e. a large number of objects live in the same root's NUMA node. On the contrary, larger rooted sub-graphs are likely to spread across NUMA nodes, especially rooted sub-graphs with size over 16K

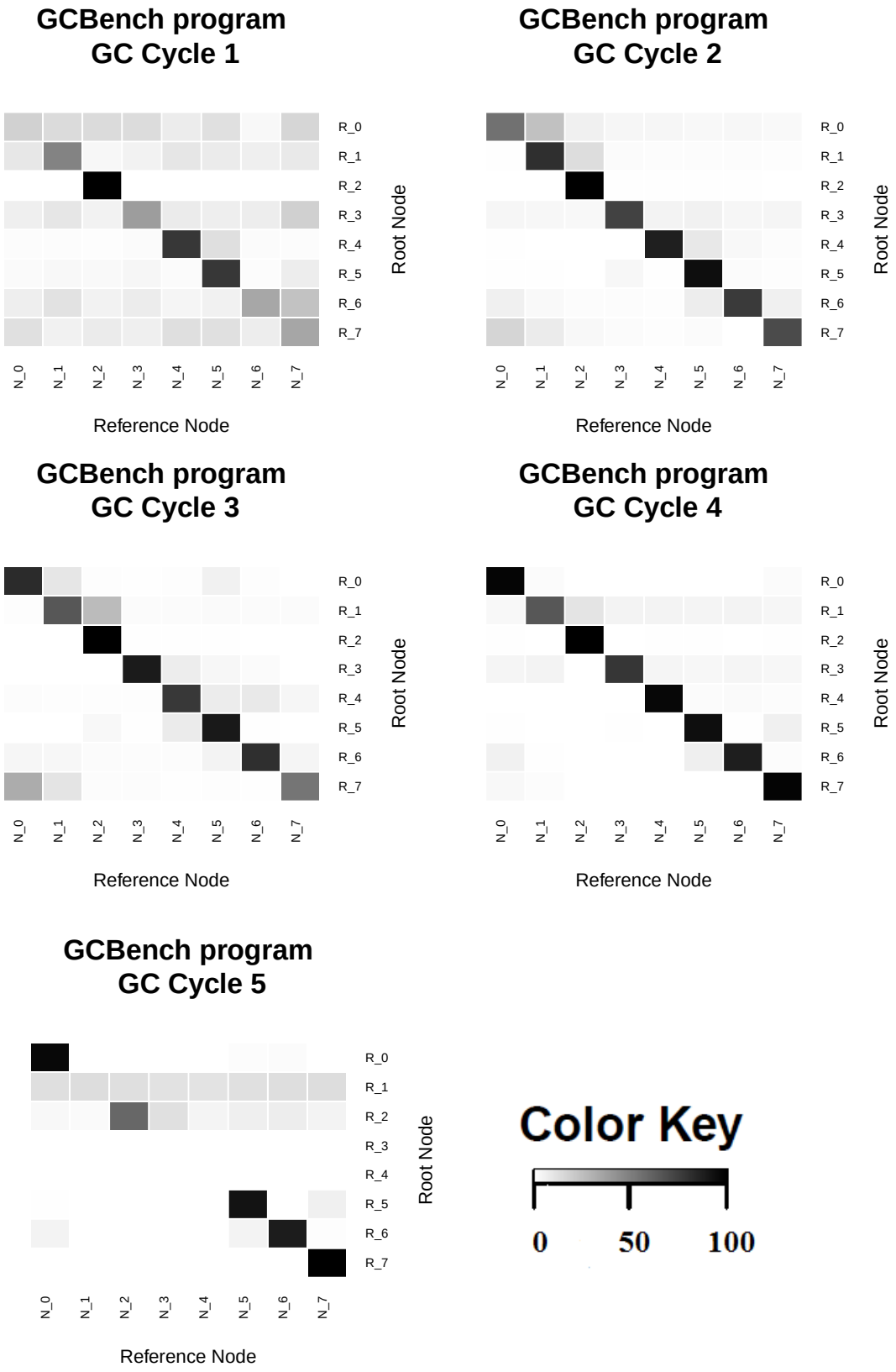


Figure 5.4: A snapshot of GCBench rooted subgraph locality in a single collection. References point to objects that are distributed across multiple NUMA nodes.

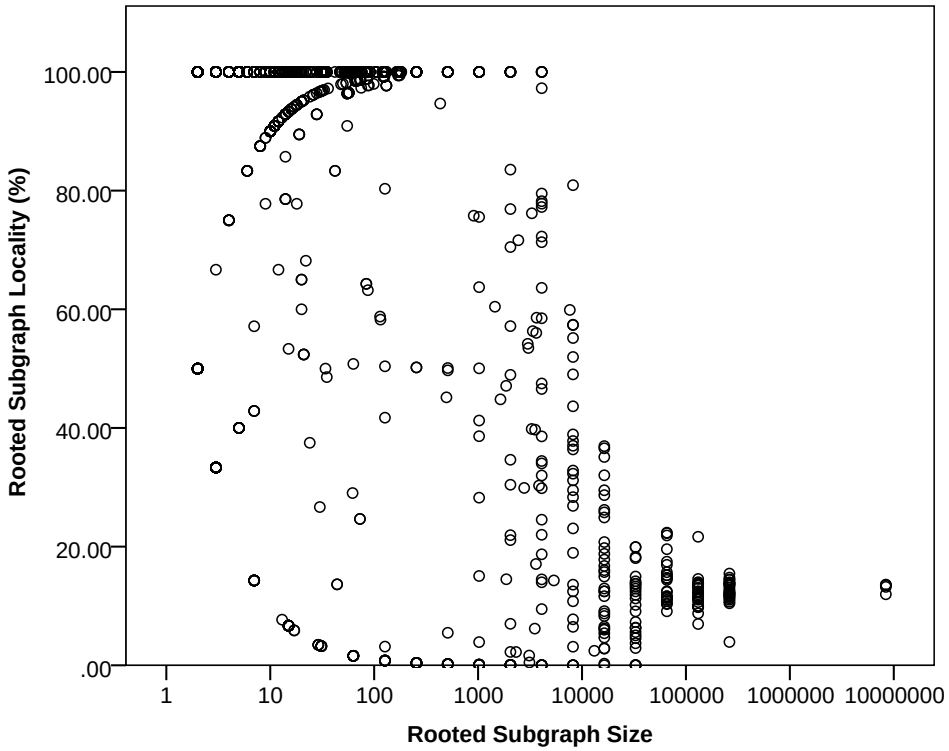


Figure 5.5: The correlation between Rooted sub-graph locality and the *size* of the sub-graph for GCBench. Large-sized sub-graphs are more likely to cross multiple memory nodes.

references. However, the proportion of large-sized rooted sub-graphs is less than 10% of the total number of rooted sub-graphs in the reference graph. Adversely, small-sized rooted sub-graphs, which provide good locality constitute to the majority of rooted sub-graphs. GCBench allocates huge memory in several phases as described in Section 4.2.3. Large-sized rooted sub-graphs span much of the program life cycle and are exposed to many garbage collection cycles. Therefore, objects would be relocated to different NUMA nodes.

The sample is quite noisy and not normally distributed; therefore, we calculate the Spearman's rank correlation coefficient. This correlation examines the monotonic relationship between the number of references in a rooted sub-graph and its NUMA locality trace. The resulted coefficient yields a moderate negative correlation (rs = -0.553), which means that as the number of references in a rooted sub-graph increases, NUMA locality trace decreases.

Next, we would explore whether certain root types contribute to form locality-distributed rooted sub-graphs. When a marker thread generates a record for every rooted sub-graph it processes, we add a field that reports the root type. The results show that there are different root types associated with poor locality rooted sub-graphs. They range from internal JVM classes and Java language classes to the running application classes. Many other application's classes and some Java language's classes contribute to form locality-distributed rooted sub-graphs. For instance, rooted sub-graphs with application class `GCBenchRunner`, and

Java language class `java.lang.ThreadGroup` creates rooted sub-graphs with references to multiple NUMA nodes.

The first collection cycle represents rooted sub-graphs locality of the mutator threads allocations in the early program's execution time. This means that the young generation space consumes memory from all NUMA nodes. Based on the transparent huge page mapping mechanism of the operating system, the Eden space and the survivor spaces in the young generation are mapped to all NUMA nodes. Therefore, objects were allocated and moved to different NUMA nodes.

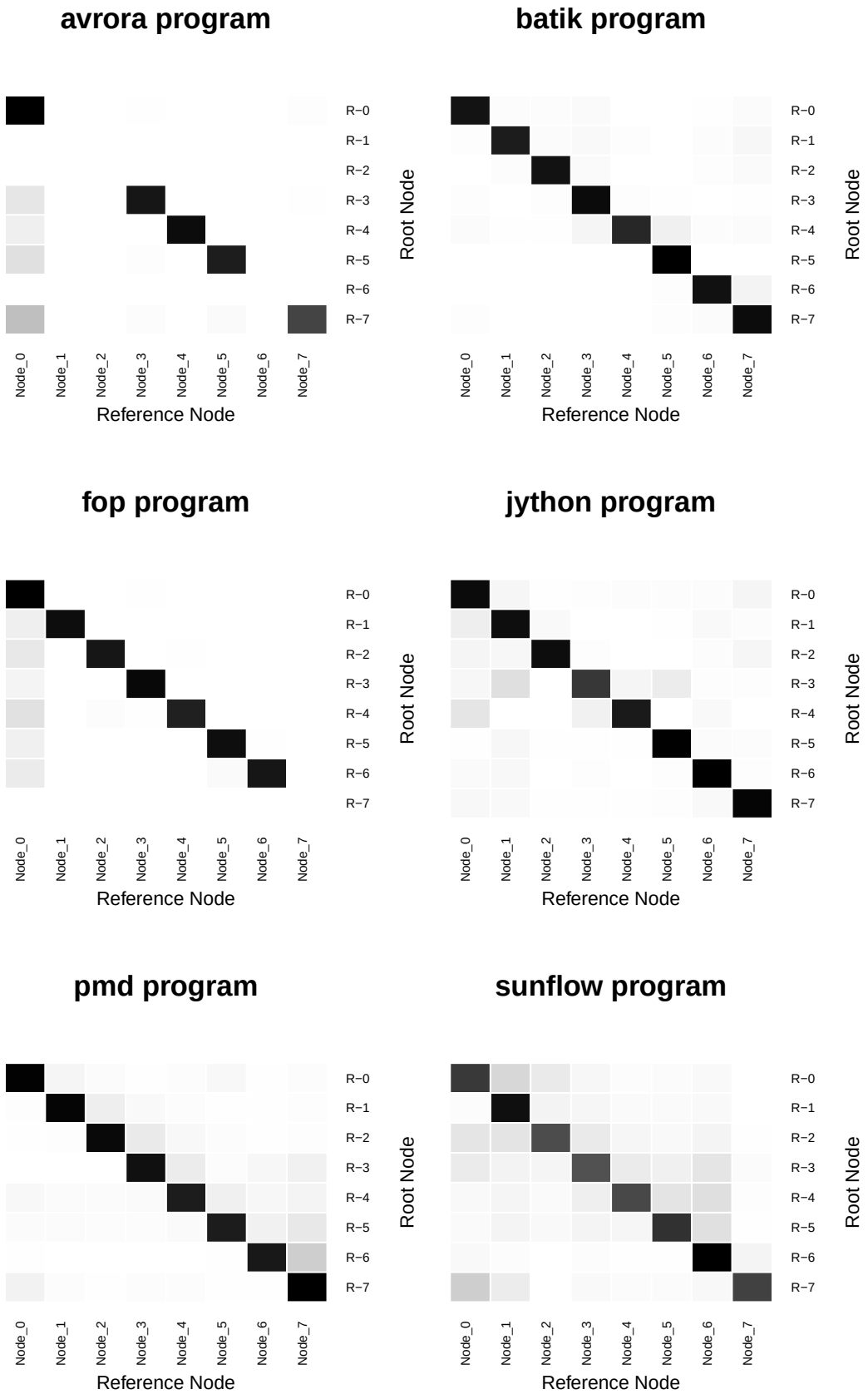
Subsequent GC cycles provide better NUMA locality trace. The major collection cycle entails a compaction operation which compacts the objects into a part of the heap. As a result, rooted sub-graphs will be packed into continuous memory pages and these memory pages are likely to be mapped to the same NUMA nodes. Accordingly, in Figure 5.4, GC cycles 2 through 4 graphs show better NUMA locality trace, which indicates that more than 60% of references in the rooted sub-graphs point to the same root node. In the last GC cycle, NUMA nodes 3 and 4 do not contain objects. We notice that the number of rooted sub-graphs has decreased, which indicates that many rooted sub-graphs have died and will be collected during the compaction phase.

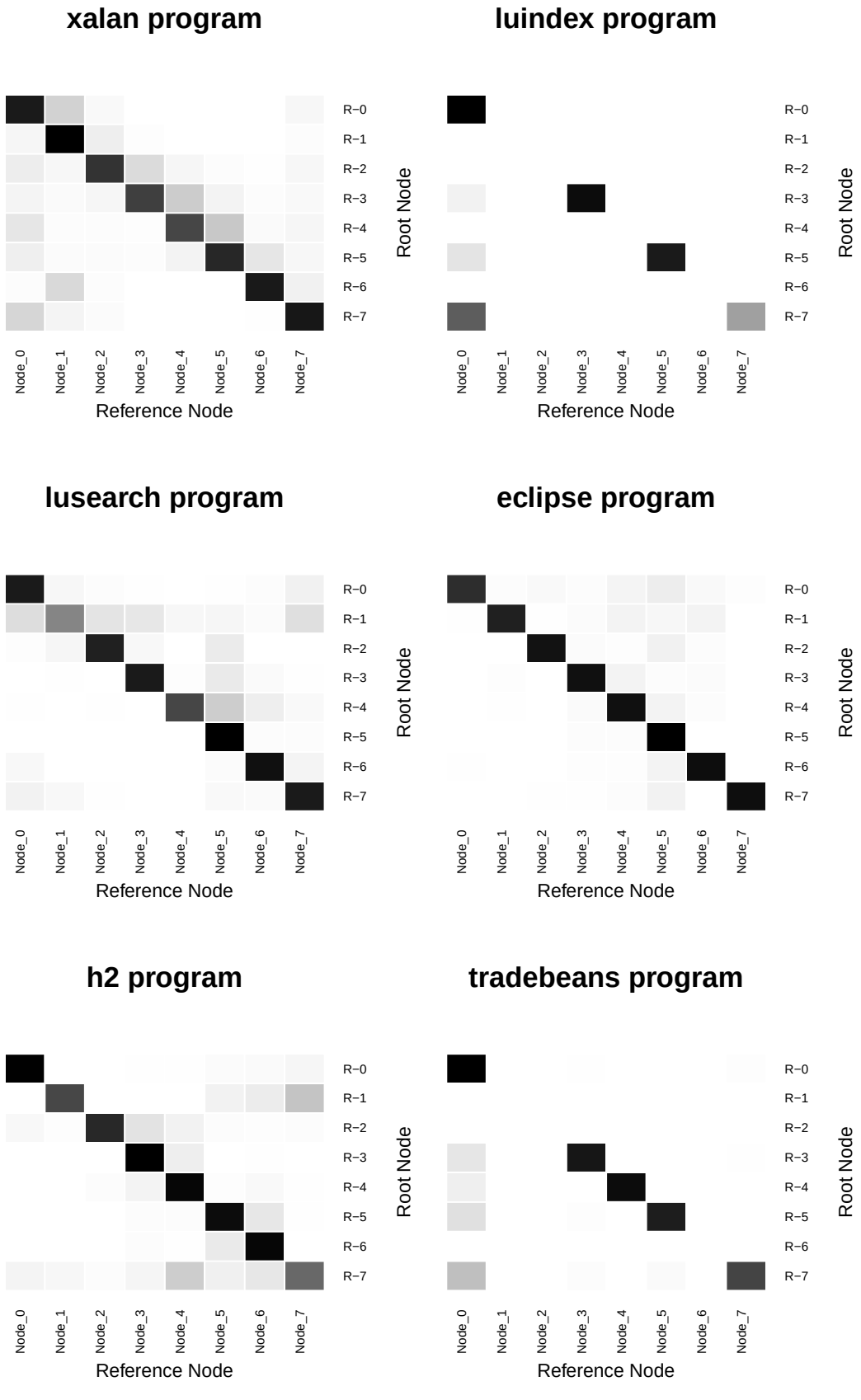
To sum up, this study gives insights on different locality forms of the rooted sub-graphs. Large-sized sub-graphs and certain root types contribute to form non-local references; whereas, small-sized sub-graphs provide good locality. Garbage collection changes the location of objects and may improve rooted sub-graph locality. Next, rooted sub-graph locality is explored for real-world applications.

5.6.2 Rooted Sub-graph Locality Analysis

In this section, we will analyze the rooted sub-graph locality of DaCapo and SPECjbb2005 benchmarks. Figure 5.6 depicts, for all GC cycles, the average percentage of references in each rooted sub-graph with respect to its root NUMA node. The results provide a strong locality relationship between references in a rooted sub-graph. This is represented by diagonal black squares, which indicate that NUMA locality trace is about 80%,i.e. objects are co-located in the same root's NUMA node.

Avrora, fop, luindex and tradebeans programs contain missing—i.e., 0%— references in some NUMA nodes. This is due to the lack of live objects in these NUMA nodes at the time of collection. Scattered sub-graphs are present in these heat maps, in particular sunflow, xalan, SPECjbb2005, and GCBench benchmarks. We would further analyze the locality results of SPECjbb2005.


avrora program
batik program
fop program
jython program
pmd program
sunflow program
R−0
R−1
R−2
R−3
R−4
R−5
R−6
R−7
Root Node
Node_0
Node_1
Node_2
Node_3
Node_4
Node_5
Node_6
Node_7
Reference Node


xalan program
luindex program
lusearch program
eclipse program
h2 program
tradebeans program
R−0
R−1
R−2
R−3
R−4
R−5
R−6
R−7
Root Node
Node_0
Node_1
Node_2
Node_3
Node_4
Node_5
Node_6
Node_7
Reference Node

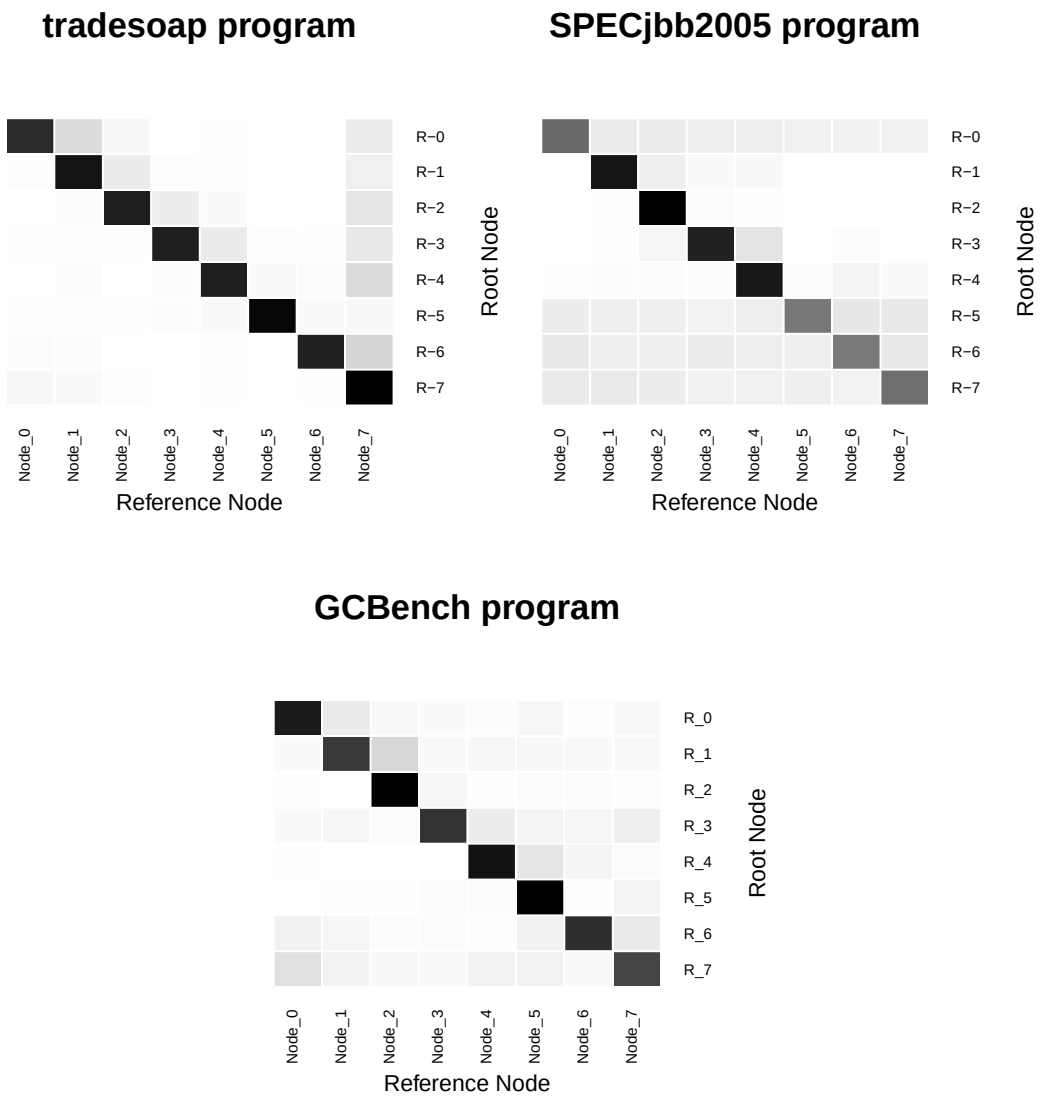


Figure 5.6: A snapshot of DaCapo and SPECjbb2005 rooted sub-graphs locality in all collections. Locality of sub-graphs is represented by diagonal black squares, which show that a high proportion of objects are located in the same root memory node. The color key is the same as in Figure 5.4.

Root references allocated in memory node 0, 5, 6, and 7 contain rooted sub-graphs with poor locality. However, 40%, on average, are allocated in the same root's memory node. Rooted sub-graphs in memory node 1 through 4 form well-organized local references with more than 80% located in the root's NUMA node.

The size attribute of locality-distributed rooted sub-graphs holds for SPECjbb2005 benchmark. It contains few rooted sub-graphs with a large number of references located in multiple NUMA nodes. A root type that contributes to locality-distributed rooted sub-graph is `spec.jbb.Transaction` class and its sub-classes. SPECjbb2005 issues many business payment and order transactions that deal with multiple warehouses using these classes. We would expect such behavior when running 64 warehouses.

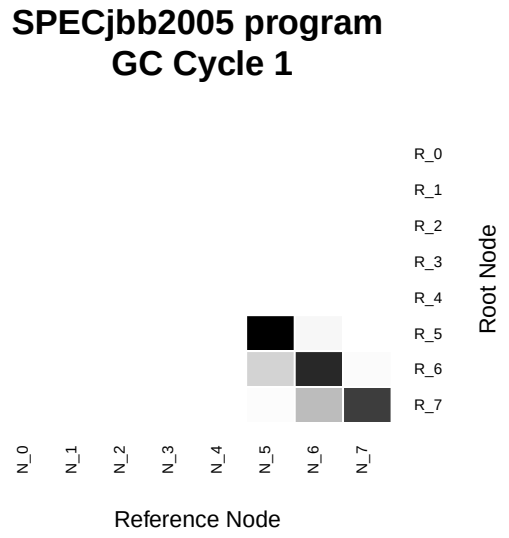
Both attributes, size and root type, influence rooted sub-graph locality. In addition, garbage collection's moving algorithms cause changes to object location. The next section discusses the impact of garbage collection on rooted sub-graph locality.

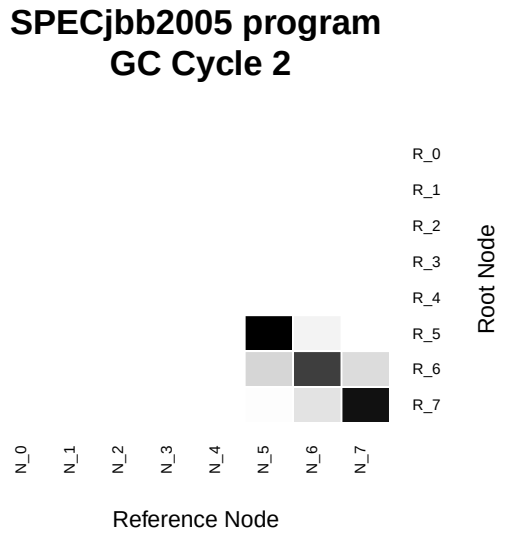
5.6.3 GC Impact on Rooted Sub-graph Locality

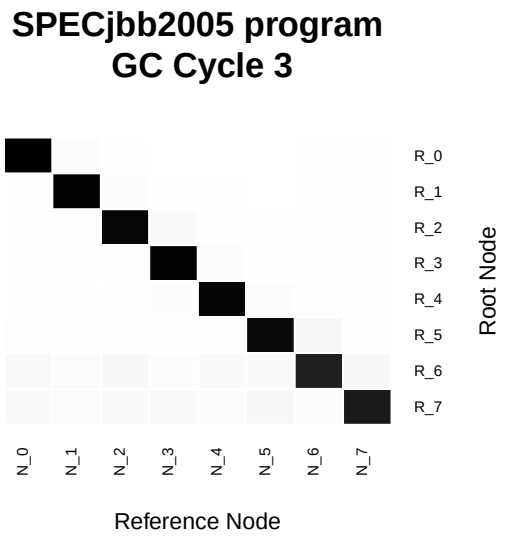
Garbage collection has an impact on rooted sub-graph locality because it moves objects around the heap during copying and compaction phases. We have observed the rooted sub-graph locality changes at the marking phase, which involves multiple copying and compaction collections. We set the heap size of SPECjbb2005 benchmark to 2.5GB, and that triggers the major collection 11 times. Figure 5.7 illustrates the GC impact on the rooted sub-graph locality.

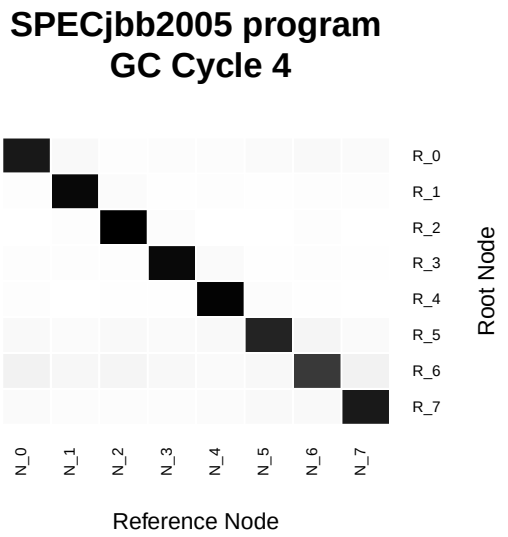
In the first two GC cycles, only three NUMA nodes participate in object allocation and evacuation from the nursery space. We analyzed the program execution and we noticed that this behavior is a result of the garbage collection's triggering time. The first two GC cycles are called in the beginning of the program run. In fact, SPECjbb2005 tests the garbage collection before running the actual program. Thus, we see that objects live in only three NUMA nodes.

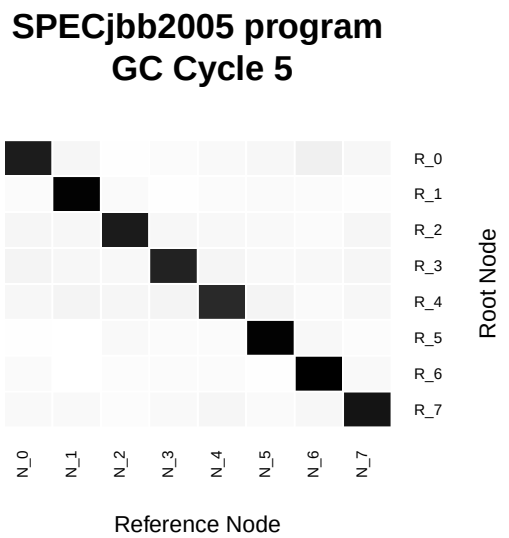
In addition, the thread mapping technique implemented by the JVM determines which NUMA nodes can be used to allocate objects. We modified the thread mapping technique in the Hotspot JVM to map the first core first, i.e., mapping threads to the first core, then the second core, and so on. This technique provides memory pages from a few nodes. For instance, an eight-thread application runs on a machine with two NUMA nodes, each of the eight cores would allocate memory from one NUMA node since the application threads will be mapped to the same NUMA node. Therefore, in early program's execution phases, a small number of mutator threads run and allocate objects, which appears to occupy three NUMA nodes.

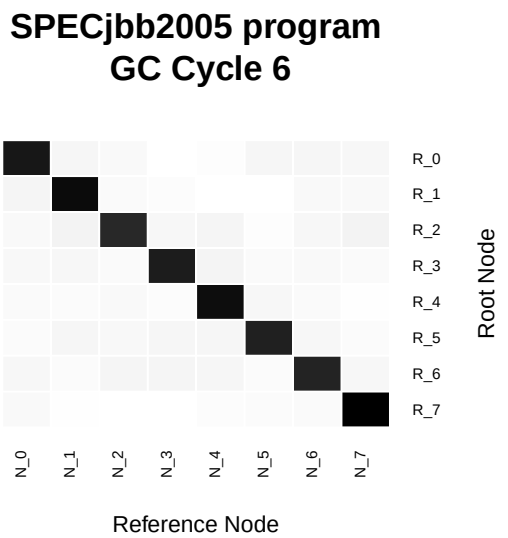

SPECjbb2005 program
GC Cycle 1
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
Root Node
N_0
N_1
N_2
N_3
N_4
N_5
N_6
N_7
Reference Node


SPECjbb2005 program
GC Cycle 2
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
Root Node
N_0
N_1
N_2
N_3
N_4
N_5
N_6
N_7
Reference Node


SPECjbb2005 program
GC Cycle 3
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
Root Node
N_0
N_1
N_2
N_3
N_4
N_5
N_6
N_7
Reference Node


SPECjbb2005 program
GC Cycle 4
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
Root Node
N_0
N_1
N_2
N_3
N_4
N_5
N_6
N_7
Reference Node


SPECjbb2005 program
GC Cycle 5
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
Root Node
N_0
N_1
N_2
N_3
N_4
N_5
N_6
N_7
Reference Node


SPECjbb2005 program
GC Cycle 6
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
Root Node
N_0
N_1
N_2
N_3
N_4
N_5
N_6
N_7
Reference Node

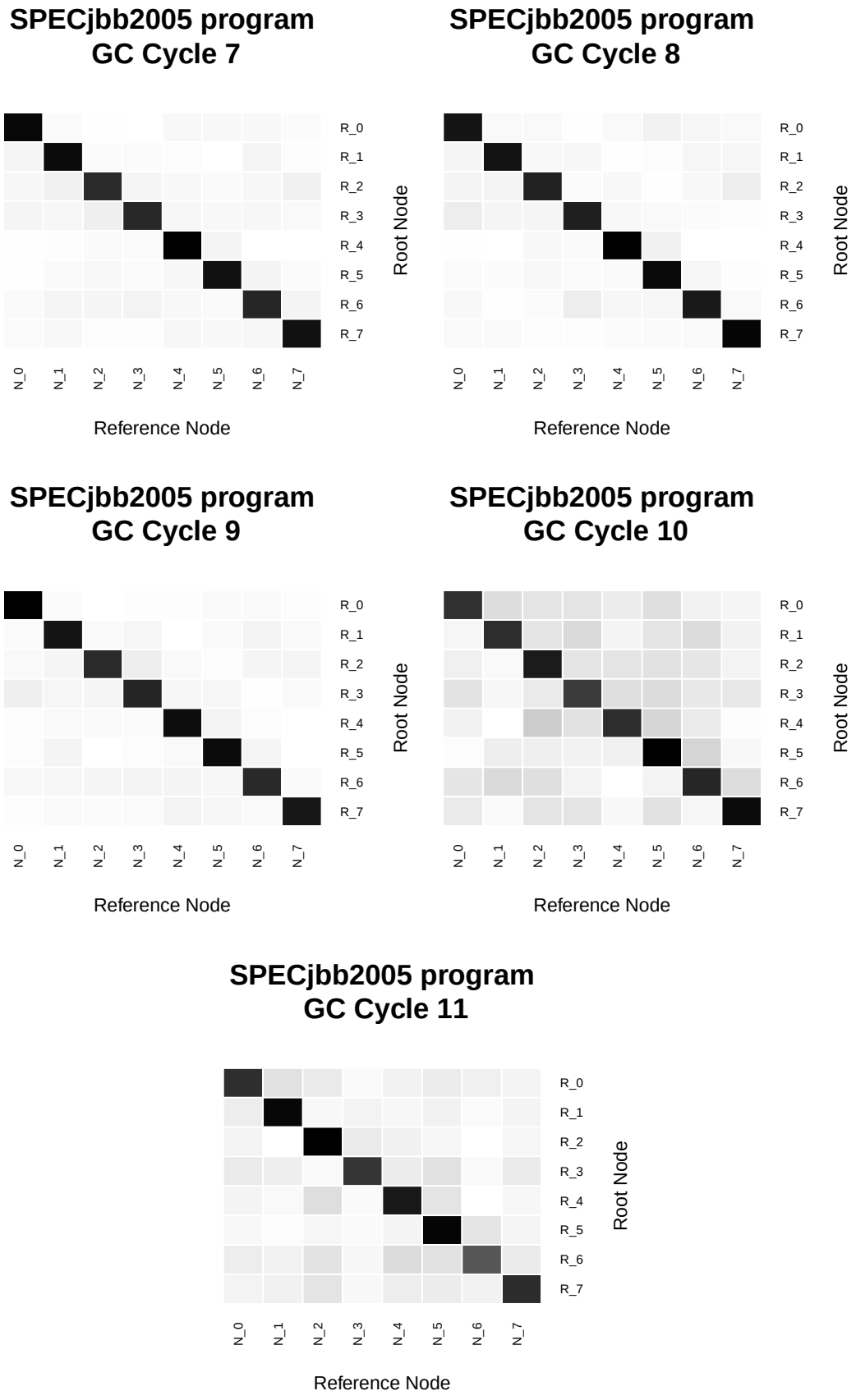


Figure 5.7: A snapshot of SPECjbb2005 rooted sub-graph locality in a single collection. The GC pollutes sub-graph locality, though over 50% of references remain in the root's memory node. The Color key is the same as in Figure 5.4.

While the program executes, the GC degrades the rooted sub-graph locality, especially in the last two cycles. There, we see gray squares around the heat map; however, there are still more than 50% of objects co-located with their root object.

Large-sized sub-graphs are amenable to work stealing; i.e. idle collector threads attempt to steal work from these sub-graphs. A stolen object and its descendant objects will be moved to the stealing thread's NUMA node. Existing work stealing implementation in the Hotpot JVM directs the stealing threads to steal from array objects as a first priority. The reason is that array objects take long time processing and include many objects. This mechanism anticipates disconnecting objects and displacing them off their original location.

From the above discussion, we can observe that the garbage collection contributes positively (e.g. compaction) and negatively (e.g work stealing) to the rooted sub-graph locality. Potential improvements would include mapping rooted sub-graph to local collector threads and restricting work stealing to local threads. By organizing the root set according to the NUMA location, GC threads would trace rooted sub-graphs with maximal locality. Moreover, restricting or setting stealing preferences of threads to acquire objects from their *sibling* threads would enhance work stealing locality. These optimizations will be investigated in Chapter 6.

5.7 Related Work

The shape of the object graph may cause problems when exploiting hardware parallelism. In particular, it may be hard to exploit deep and narrow data structures—for example linked-lists and arrays [Barabash and Petrank, 2010, Eran and Petrank, 2013]. Such data structures lead to an unbalanced load assigned to parallel threads. We have shown that deep rooted sub-graphs contain locality-distributed references to multiple NUMA nodes. Thus, tracing deep rooted sub-graphs would impact the garbage collector performance due to remote memory access overhead.

Several optimizations have been provided to improve the parallel garbage collection tracing techniques. Processor-oriented techniques attempt to balance the load over garbage collector threads and keep them busy tracing live objects. Endo et al. [Endo et al., 1997] develop a parallel mark-sweep garbage collection algorithm with work stealing technique. A collector thread fills an auxiliary queue to enable idle threads to steal half of the work. Flood et al. [Flood et al., 2001] implement parallel garbage collectors with statically over-partitioned root tasks as well as work stealing technique. The granularity of work stealing is set to the object size. These optimizations provide arbitrary work partitioning that discard object locality. However, we propose a partitioning scheme in which root references are clustered according to their NUMA node and direct garbage collector threads to consume their corresponding queues.

In contrast, memory-oriented optimizations focus on improving object locality by tracing per-memory-region objects. Memory is segregated into segments and each segment is associated with a work list of local objects only. Remote objects are transferred to a shared queue or to the local queue of the remote region. Shuf et al. [Shuf et al., 2002a,b] exploit the prolific types placement technique and collect local objects in each memory region. Garbage collector threads insert remote objects in a shared work list, and provide work stealing at object granularity. To reduce shared work list synchronization, Chicha and Watt [Chicha and Watt, 2006] transfer remote objects to their respective region. Oancea et al. [Oancea et al., 2009] partition the memory and create a work list per region. A processor must own a work list to trace the local objects; consequently, load balancing is achieved by stealing a *complete* work list. The locality improvement in this work is associated with local objects in each heap region; however, object connectivity may be changed if objects are processed by threads mapped to remote NUMA node. We attempt to exploit rooted sub-graph locality and propose partitioning the root set and directing garbage collector threads to the appropriate queue. For work stealing technique, we propose restricting the garbage collector threads to steal from *sibling* cores to benefit from local memory access and to preserve rooted sub-graph locality.

The order of tracing the object graph is important to object locality [Jones et al., 2011]. Objects traversed in breadth-first order—for example Cheney algorithm [Cheney, 1970a]—tend to be separated from each other. In contrast, depth-first traversal order provides better locality since it co-locates its parent and its children objects [Wilson et al., 1991]. Other techniques attempt to group objects based on function order such as creation order, that is, the time objects being created [Wilson et al., 1991]; hierarchical decomposition order based on object types [Lam et al., 1992]; or online object reordering that copies hot fields of objects with their parents [Huang et al., 2004]. These researchers attempt to increase object locality for optimizing processor cache access time— i.e. improve spatial locality. Similar techniques may be applicable to improve NUMA locality.

Locality optimizations for mutator threads attempt to improve data placement techniques. Local and newly instantiated objects are allocated in thread-specific heaplets since they are, with high probability, accessed by the thread itself [Domani et al., 2002, Steensgaard, 2000, Marlow and Peyton Jones, 2011, Anderson, 2010, Jones and King, 2005, Auhagen et al., 2011]. Furthermore, heaplets can be collected independently since objects are local and there is no need to synchronize all mutator threads. Shared objects are allocated in the NUMA node of the thread accessing them the most, see Tikir and Hollingsworth [2005], Ogasawara [2009]. We observed that without NUMA awareness, garbage collection would change the object locality.

Zhou and Demsky [Zhou and Demsky, 2012] implement a master-slave paradigm for mutator and collector locality optimization. Every thread has its private heap space for allocation

and collection. At collection time, if there is any pointer to non-local object, the garbage collector thread sends a message to the master thread and instruct the target thread to process that object. They manage the cost of message passing by hardware support; otherwise, it could harm the performance [Gidra et al., 2013]. While this study attempts to improve object locality, Majo et al. [Majo and Gross, 2011] suggest that accessing remote memory could reduce cache contention in local NUMA nodes. Rooted sub-graphs in our benchmarks provide evidence that they combine local and remote objects such that they can be used to balance remote and local memory accesses.

Object connectivity has been studied to explore object lifetime property. Hirzel et al. [Hirzel et al., 2002] explore various kinds of connectivity and analyze their correlation with object lifetime and deathtime. They suggest that placing connected objects close to each other is beneficial because they have similarities in deathtime. We observe object connectivity from a locality aspect and find that rooted sub-graphs have a strong correlation with locality.

5.8 Conclusion

This chapter has demonstrated the advantage of partitioning the reference graph into rooted sub-graphs. The memory affinity relationship between a root reference and its reachable set, which is represented here by rooted sub-graph, tend to be strong. A high percentage of references in a rooted sub-graph reside in the same NUMA node as the root.

Constructing rooted sub-graphs should not incur additional computational or space overhead. The reference graph that the garbage collector traverses supports reference discrimination between roots and their transitive closures. Our hypothesis emphasises that by preserving this reference discrimination, garbage collector would be able to create rooted sub-graphs and preserve the intrinsic locality.

Huge transparent pages, as supported by some operating systems (Section 3.5), are a key factor in increasing rooted sub-graph locality. A majority of rooted sub-graphs can fit in a single huge memory page. With both reference graph partitioning and huge transparent pages, the garbage collector can benefit from locality gains available in the rooted sub-graphs.

To test rooted sub-graph locality, we conducted an empirical study on DaCapo benchmarks as well as GCBench, which is a micro benchmark designed to demonstrate cross-node reference impact. Since the system's memory size is huge (512GB), we would expect no effect from memory swap or shortage when other processes run on the same time, though the experiments were performed while there is minimal background load on the system.

Our evaluation suggests that, on average, NUMA locality trace of rooted sub-graph is 80% with 7.9% error rate, which means that 80% objects in a rooted sub-graph are allocated in

the same NUMA node as the root. We analyse two factors for locality-distributed rooted sub-graphs: size and root type. We find that locality-distributed rooted sub-graphs tend to be large in size; thus, become exposed to load balancing parallel techniques, i.e. work stealing. Stealing an object from a rooted sub-graph may result in placing it—and possibly its descendants— into a remote NUMA node. In addition, some internal JVM and application classes contribute to create locality-distributed rooted sub-graphs. These two factors could help programmers and developers to deal with peer locality of references they create.

We conjecture that there may be potential benefits when using the root's NUMA node as a locality heuristic to improve garbage collection performance. In the next chapter, we will modify the Hotspot JVM to take advantage of the rooted sub-graphs locality heuristic.

CHAPTER

6

NUMA-AWARE GARBAGE COLLECTOR

The intrinsic locality of rooted sub-graphs, which is discussed in Chapter 5, can be utilized to improve the garbage collector performance. When a garbage collector thread traces a rooted sub-graph, it is likely to spend most of its tracing time in a single NUMA node. This chapter aims to implement a NUMA-aware garbage collector. This implementation consists of three stages. First, a collector thread scans a root area where root references are located, classifies roots with reference to their NUMA node, and pushes them into the corresponding shared NUMA queue. Second, a collector thread acquires a root reference from a NUMA-local queue and traces its transitive closure. Third, for work stealing, a collector thread looks for work from pending queues in the local NUMA node.

This chapter makes the following contributions:

1. It modifies a stop-the-world copying collector algorithm and makes it NUMA-aware.
2. It proposes various optimization schemes that take into account NUMA topology in task processing and work stealing.
3. It measures the performance gains of the optimization schemes by examining a range of workloads that span a wide spectrum of heap sizes.

6.1 Introduction

The parallel techniques incorporated in collecting the heap focus on generating, partitioning, and stealing tasks to speed up collection. Many of these techniques are designed with temporal locality considerations. For moving garbage collection algorithms, e.g. copying and compaction garbage collection, object are co-located in the same virtual memory page to improve locality. Object co-location is performed by various graph traversal techniques, for example depth-first, see Section 2.5. However, these techniques may be not sufficient for NUMA systems. Heap allocated objects may be placed closely in the virtual memory pages but not necessarily in physical memory.

At a higher program execution layer, many operating systems, for example Linux and BSD, support a huge memory address space. In such a situation, a large TLB buffer size is required for paging. To reduce TLB buffer size and improve its performance, huge page tables are used, *more details in Section* 3.5.

Hardware and software techniques for tackling specific issues may cause further problems when they are integrated together. JVMs and other runtime systems, for example Common Language Runtime (CLR) which is the virtual machine for .NET framework, abstract such low-level platform-specific details. In addition, evolving technologies and diversity in hardware deployments as well as rapid development efforts in operating systems may hinder runtime systems from gaining expected advantages, if these technology advancements are not successfully integrated. Devolving some virtual machines' services such as memory allocation management to the operating system might not be effective as described in Section 3.4. Moreover, cache coherency for NUMA systems suffers from false sharing problem, where multiple cores share a cache line without sharing data, thus it impacts multi-threaded application performance [Berger et al., 2000]. Therefore, future systems may not employ cache coherency protocols, and virtual machines should response to such changes, see Section 8.3.2.

NUMA systems distribute memory banks across processors and connect them via high speed links to improve memory access bandwidth. A multi-threaded application running on a NUMA machine may place data in any NUMA nodes; thus threads may need to access remote memory. In this architectural layout, individual processor cores are likely to incur non-uniform memory access latency. For non NUMA-aware managed runtime systems, multi-threaded applications may exhibit unpredictable and suboptimal application performance.

Previous research pays considerable attention to improve garbage collection locality by allocating data close to the core most frequently accessing it Ogasawara [2009] and Gidra et al. [2015]. However, such data placement policies could conflict with NUMA's intrinsic design

(distributed memory) because locality may direct traffic to some NUMA nodes and cause their interconnection links to saturate. Furthermore, improving locality may stress the local cache hierarchy, while off-node resources could provide abundant memory capacity. Therefore, allocation balance is significant for NUMA architectures, [Dashti et al., 2013, Majo and Gross, 2011]. There should be a technique that balance the need for improving data locality while maintaining available resources in the system.

Chapter 5 has shown that there is an opportunity to improve garbage collection locality by utilizing the rooted sub-graph hypothesis. By exploiting huge page tables, which co-locate objects in the same NUMA node, the work scope, i.e. individual rooted sub-graphs, for a garbage collector thread is likely to be within a single NUMA node.

This chapter proposes three optimization schemes to improve garbage collection performance (Section 6.3.3). First, the aggressive scheme enforces collector threads to process local rooted sub-graphs only. Second, the hybrid scheme enables work stealing threads to steal from off-node rooted sub-graphs. Third, the relaxed scheme allows collector threads to process any reference from any rooted sub-graph.

The copying collector of the Hotspot JVM is modified to implement a NUMA-aware parallel copying collector. This algorithm is evaluated with various workloads, see Section 4.2. The results show that leveraging rooted sub-graph's locality characteristic improves substantially of the garbage collection performance (**15%**) on average.

6.2 Motivation

Hotspot JVM uses three-phase garbage collection in the existing Parallel Scavenge policy, see Section 3.6. The copying collector incorporates conventional techniques to manage the parallel phase. The following list describes these techniques and pinpoint potential optimizations, where we can gain better performance using the rooted sub-graph hypothesis 5.2.

1. **Root classification:** the serial phase of the copying collector prepares a list of memory areas where root references can be found, for example static areas, mutator stacks, and JNI handlers. Each collector thread processes at least one memory area. The discovered root references are enqueued in a local queue of the task-generating thread. Memory areas may have different number of roots. In this design, whenever a collector thread completes processing its rooted sub-graphs, it attempts to steal from a non-empty queue to balance the load.
   Optimization: instead of entering the work stealing phase, stealing a root is better than stealing a descendant reference which would destroy sub-graph integrity. Thus,

this research proposes to enqueue root references in shared queues to allow collector threads to process roots before stealing non-root from other queues.

2. **NUMA-local reference processing:** each collector thread enqueues root references in a thread-local queue. These root references may refer to objects in any NUMA node; thus, a collector thread could process a remote sub-graph.
   Optimization: We propose creating a per-node shared queue to store roots. Each task-generating thread classifies discovered root references and pushes them into appropriate NUMA queues.

3. **NUMA-local work stealing:** after processing all the references in the local queue, a collector thread selects a random pending queue and steals a single reference.
   Optimization: to improve locality, we propose stealing only from NUMA-local thread queues with reference to the underlying NUMA topology. In this proposal, a stealing thread avoids crossing off-node links to steal a remote reference, which could have further references to remote NUMA nodes.

The Parallel Scavenge policy devotes its design to keep the collector threads busy collecting the heap without considering the complex NUMA architecture. Although the *optional* NUMA configuration can help garbage collection to be aware of NUMA topology, its implementation has been reported as ineffective [Gidra et al., 2013], see Section 3.6.3 page 53. In addition, the garbage collection may change object location, (during a copy-promotion), and relocate it to a different NUMA node; hence mutator threads would incur remote accesses and degrade application performance. The proposed optimizations aim to avoid these problems.

The rooted sub-graph hypothesis is the basis for the proposed optimizations. Previous research for NUMA copying collector apply optimizations at a per-object granularity of work. Tikir and Hollingsworth [2005] profile memory access patterns for each object and calculate the target NUMA node as the core accessing an object most of the time. Ogasawara [2009] traces a sub-graph and moves objects to the *dominant thread's* node. The dominant thread is the thread that is likely to access the object most frequently.

However, rooted sub-graphs exhibit abundant locality, where there is a large proportion of objects in a sub-graph co-located in the same NUMA. We have shown that choosing rooted sub-graphs as the work granularity for task generation and distribution on NUMA machines can yield better collection performance.

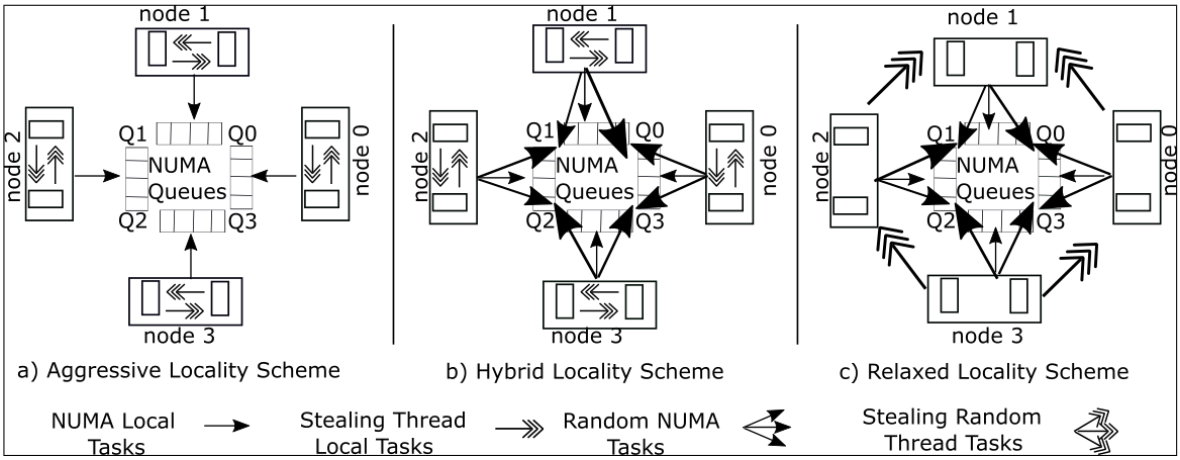


Figure 6.1: Various topology-aware GC schemes. a) aggressive scheme only processes thread-local tasks b) hybrid scheme distributes tasks across all nodes but steals from local threads only. c) relaxed scheme processes random tasks from any node

6.3 NUMA-Aware Copying Collector

This section presents the design and implementation of the proposed NUMA-aware copying collector. Before delving into details, we highlight the following considerations from Section 6.2. First, NUMA awareness in our algorithm is centred at preserving the rooted sub-graph integrity, which means that connected objects in the sub-graph are processed together. The Hotspot JVM does not differentiate between root reference and other references. In our algorithm, we separate the task processing into task generation and task distribution. As a result, garbage collector threads consume the root list first before entering the work stealing phase; hence they would copy the entire rooted sub-graph to the same physical location.

Second, garbage collection threads should process *local* root references. Local roots are identified during task generation by classifying the root set according to NUMA nodes. Therefore, a collector thread dequeues a root reference from NUMA-local queue.

Third, when a collector thread exhausts its local work queue, it enters the work stealing phase. To enable low access latency in this phase, work-stealing threads should search for references from non-empty queues of *sibling* cores, i.e. in the same NUMA node. Garbage collection threads that are running on the same NUMA node benefit from shared resources (e.g. caches). Furthermore, stolen objects will be copied to the same NUMA node of their original sub-graph. Therefore, the locality is preserved.

6.3.1 Data Structures

Figure 6.1 illustrates the data structures used in our NUMA-aware copying collector. When the JVM launches, we query the operating system to discover the NUMA topology. At this

stage, we create as many NUMA queues as the underlying NUMA nodes. Garbage collection threads run concurrently; thus we need to facilitate thread safe enqueue and dequeue operations. Therefore, we utilize the existing Hotspot `Arora` queue data structure, which implements the single producer / multiple consumers paradigm. At one end of the queue, garbage collector threads dequeue root references safely using atomic operations. At the other end, we protect the queue with a lock such that garbage collection threads must acquire the lock first to perform an enqueue operation.

There could be lock contention on NUMA queues because garbage collection threads are concurrently trying to enqueue tasks. Therefore, we *mirror* these shared NUMA queues for each garbage collection thread such that roots are buffered locally. Once a root scanning task is completed, a garbage collection thread locks the appropriate NUMA queues (one at a time) and flushes the corresponding local NUMA queues into the shared queues. Although we create many queues, the memory footprint remains small because the root set size is small compared to the live object set.

Since some garbage collection threads may complete root scanning tasks early, it is possible that a garbage collection thread might attempt to dequeue a root reference from the corresponding shared NUMA queue but it might fail because there are no roots enqueued yet. As a result, a garbage collector may enter the work stealing phase whilst other collector threads still scanning the root areas. In order to prevent collector threads from entering the work stealing phase so early, we set a threshold for the mirrored NUMA queues. A collector thread that buffers a certain number of root references should transfer all discovered references to the corresponding should NUMA queues. We set the threshold to 100 references.

6.3.2 Algorithm

When program execution pauses for garbage collection, a single thread calls the VM thread that executes a sequential block of code. In preparation for the parallel phase, the VM thread populates the `GCTaskQueue` with three types of tasks, see Section 3.6 for more details.

1. *Root scanning* task: These task scans various memory areas to discover root references.

2. *Stealing* task: Threads that run out of work steal references from pending queues to balance the load.

3. *Finaliser* task: These task manages the termination of the parallel phase.

We modified the root scanning task to include proposed task generation and task distribution of root references. Initially, every garbage collection thread scans one or more memory

areas for root references. A garbage collection thread obtains the NUMA node of every root reference and pushes it in the corresponding thread-local queue. If any local queue reaches the size threshold, it tries to lock the shared NUMA queue and transfer all references. Once a collector thread finishes root scanning tasks, it starts processing root references. It acquires a root reference using atomic operations and traces its reachable set. At this stage, garbage collection threads use a thread-local queue to trace the reachable reference set as in the default implementation. Once the root set is processed, garbage collection threads execute stealing tasks and search for references from pending queues. The first thread that finishes its work stealing phase executes the finalizer task to manage the parallel phase termination. Our algorithm's pseudo code is presented in Listing 6.1.

Listing 6.1: NUMA-Aware Copying Algorithm Pseudo Code

```
Task = acquire_gc_task()
switch (Task)
case scan_roots:
   for(all_root_areas){
      root = discover_roots()
      node = retrieve_root_node(root)
      enqueue_local_queue(root, node)
      if(queue(node)_size()>threshold)
        for(i =0; i<threshold; i++)
           enqueue_Arora_queue(root, node)
   }
   break;
case steal_work:
   node = get_thread_node()
   while(Arora_queue(node) != empty){
      ref = dequeue(node)
      follow(ref)
   }
   while(NUMA_local_queue(node) != empty){
      ref = dequeue()
      follow(ref)
   }
   break;
case final_task:
   wait_until_all_threads_terminate()
   hand_control_to_VM_thread()
   break;
   end
```

Work Stealing	Task Working	
	Local	Non-local
Local	Aggressive	Hybrid
Non-local	N/A	Relaxed

Table 6.1: Optimization schemes for NUMA-aware garbage collection.

6.3.3 Optimization Schemes

Our NUMA-aware garbage collector considers the NUMA topology at various stages, most importantly, the task distribution and work stealing. Classifying object's NUMA node requires system calls using the *libnuma* library [Andreas, 2005]. However, this system call is expensive and if we use it, the overhead will be proportional to the size of live object set. In addition, data locality optimization may cause NUMA congestion in the local memory controller or links. Therefore, we explore three optimization schemes to study the impact of applying data locality and/or NUMA memory balance. In all cases, rooted sub-graph integrity is preserved, i.e. garbage collection threads process root references first.

Since this study has two factors: data locality and NUMA memory balance applied on task working and work stealing, we should test four possibilities:

Aggressive: garbage collection threads look up object's NUMA node at task generation phase, and only steal references from NUMA-local threads as described in Section 6.3.1 and Section 6.3.2.
Hybrid: garbage collection threads process roots randomly; however they steal from sibling (NUMA-local) queues only.
Relaxed: garbage collection threads process roots randomly and steal work from any queue. We exclude the option: local roots/random stealing because the combination of system calls and random stealing produces huge overhead; hence degrades the garbage collection performance. Table 6.1 lists the three optimization schemes and Figure 6.1 depicts a schematic overview for them.

6.4 NUMA-Aware Garbage Collector Evaluation

This section evaluates our three optimization schemes. First, we describe various evaluation metrics: object locality, execution time, and scalability. Next, we analyze and discuss the results. Experimental setup and benchmark workloads are described in Chapter 4.

6.4.1 Evaluation Metrics

We use three different metrics to evaluate our optimization schemes.

NUMA Locality Trace:

In Chapter 5, we have studied object locality empirically and used Relative NUMA locality Trace as a measure for object locality, see Chapter 5.5 page 70. The study focused on objects in the old generation using OpenJDK version 7. This study evaluates object locality in the young generation because our optimizations are applied to the copying collector. In addition, we use OpenJDK version 8 with an up-to-date change set at the time of the study. We expect that there would no major difference in object locality between these two collectors except that the copying collector has an additional root area, which is the references from the old to the young generation. Section 6.4.2 discusses this issue.

Application Pause Time and Total Execution Time

We measure and report the pause time caused by the (stop-the-world) garbage collection and the end-to-end execution wall-clock time of the JVM. Time measurements are taken five times. We report arithmetic means, and plot 95% confidence intervals on graphs.

Scalability

In this study, we schedule as many collector threads as the number of cores available to the system. However, workloads that require large heaps may incur a scalability bottleneck. The copying collector scans the old generation for roots that have references to the young generation. Usually, in generational heaps, the garbage collection uses a card table, which is a data structure used to record old-to-young pointers. As the old generation heap size increases, the card table grows as well and collector threads consume much time for root scanning. Therefore, we investigate the responsiveness of our optimization schemes to the changes of heap size.

6.4.2 Relative NUMA Locality Trace

Figure 6.2 shows the relative NUMA locality trace results. Due to the huge trace file size for Neo4j / LiveJournal, we are unable to process all the data collected. However, we follow a previous research practice ([Gidra et al., 2015]) and limit our study of this benchmark to the fifth collection cycle only as a sample of the workload collection phase.

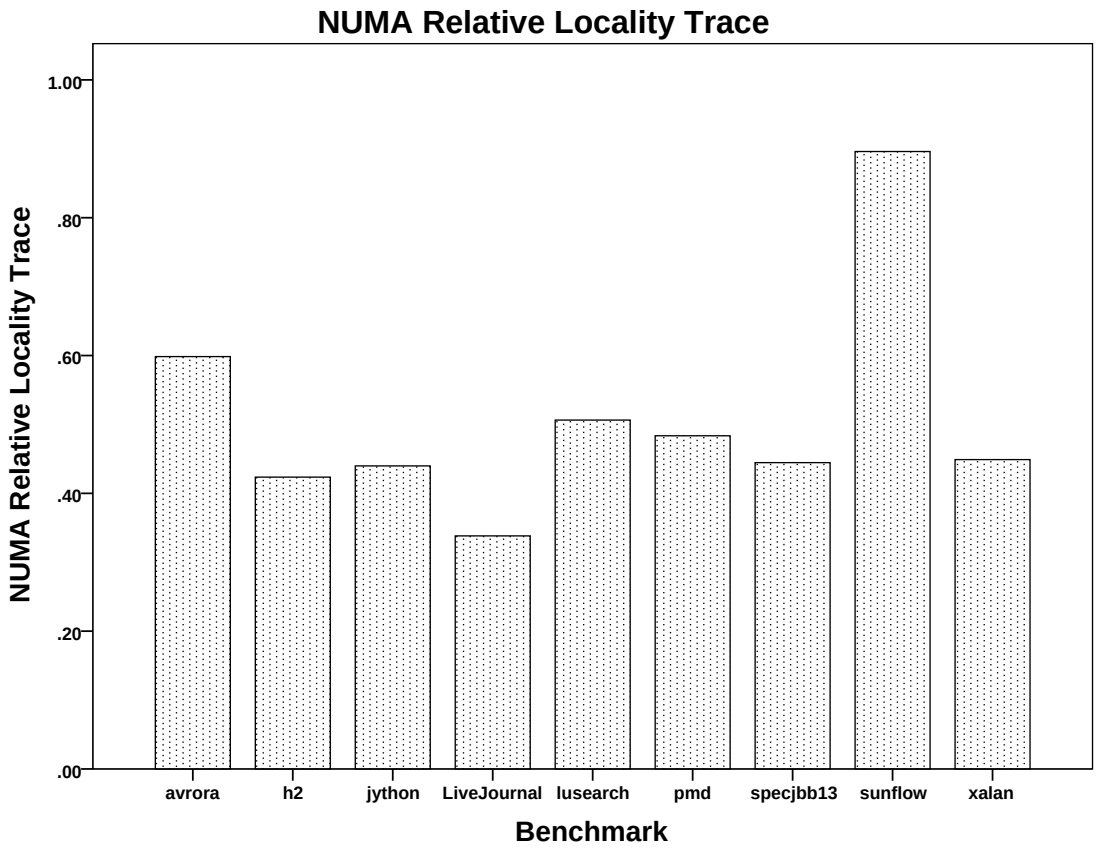


Figure 6.2: Relative NUMA Locality Trace results for evaluated workloads. On average, 53% of objects are NUMA-local within rooted sub-graphs.

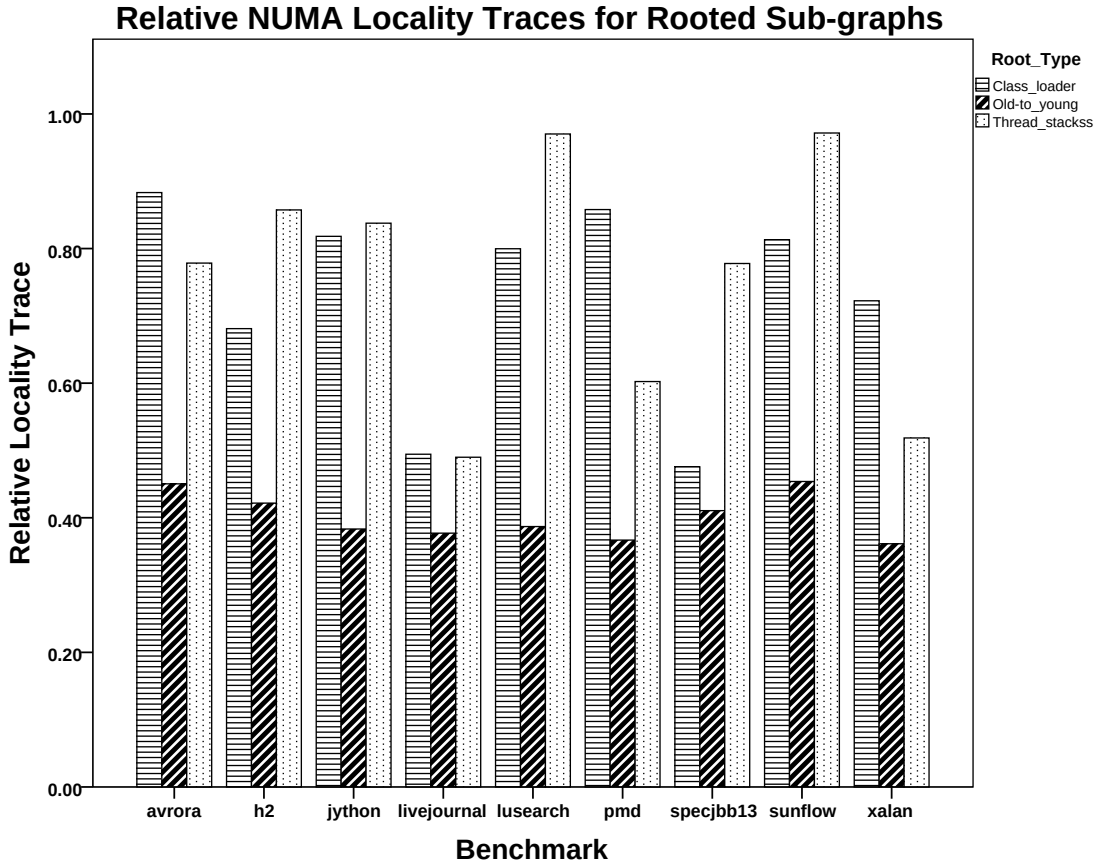


Figure 6.3: Relative NUMA Locality Traces for various root types: old-to-young, thread stacks, and class loader roots. *Old-to-young rooted sub-graphs exhibit relatively low locality*.

DaCapo/Sunflow obtains the best relative NUMA locality trace results. Approximately, 90% of the application's objects reside in the same node as the root. On the contrary, DaCapo/h2 shows the lowest relative NUMA locality trace results from DaCapo benchmark suite. Objects in DaCapo/h2 are distributed across NUMA nodes and only 42% of objects in rooted sub-graphs live in the same NUMA node as the root. Neo4j/LiveJournal records the minimum relative NUMA locality trace values with 35%. We can notice that our optimization schemes results exhibit , in general, less variation. This less variation is due to the work unit, i.e. rooted sub-graphs, we set for garbage collection threads, which improves object locality. The main difference between DaCapo benchmarks and LiveJournal program is the heap size. We will discuss how the heap size affects rooted sub-graph locality in the subsequent sections. For all workloads, the relative NUMA locality trace is 53% on average.

We can notice that results differ from our earlier empirical study in Chapter 5, which has demonstrated high rooted sub-graph locality. This is expected because the copying collector differs from the mark-compact collector in the number of memory areas included in the task generation phase. When collecting the young generation, the garbage collector includes the card table to scan references from the old to the young generations. Consequently, these results may suggest that we cannot rely on the locality characteristic of rooted sub-graphs to optimize the copying collector. However, we run more experiments to investigate whether there is any factor that gives more insight on rooted sub-graph locality changes.

Recall that at the beginning of parallel garbage collection phase, several root scanning tasks are inserted in the shared queue `GCTaskQueue`. Root scanning tasks direct the collector threads to various JVM data areas, where potential root references can be found. These memory areas include but not limited to mutator stacks, card table (for old-to-young references), JNI handlers, and class loader data. We assume that some of these root types may dominate the root set, and their relative NUMA locality trace can affect the overall results. This assumption is similar to the prolific type notion study, see [Shuf et al., 2002a]. Therefore, we calculate relative NUMA locality traces for prevalent root types and plot the results in Figure 6.3. For all evaluated workloads, the old-to-young rooted sub-graphs consistently obtain low locality results, whereas other roots show high locality.

These results suggest that our optimization schemes can be applied on high-locality root types. In the next section, we show that garbage collection performance increases only when applying locality optimization on *all root types except old-to-young references*. For old-to-young root, we randomly assign root references to any NUMA queue.

6.4.3 Pause Time and VM Time Analysis

Figure 6.4 and Figure 6.5 plot the garbage collection pause time and VM execution time results for our workloads. A boxplot shows the median, the first and the third quartile and

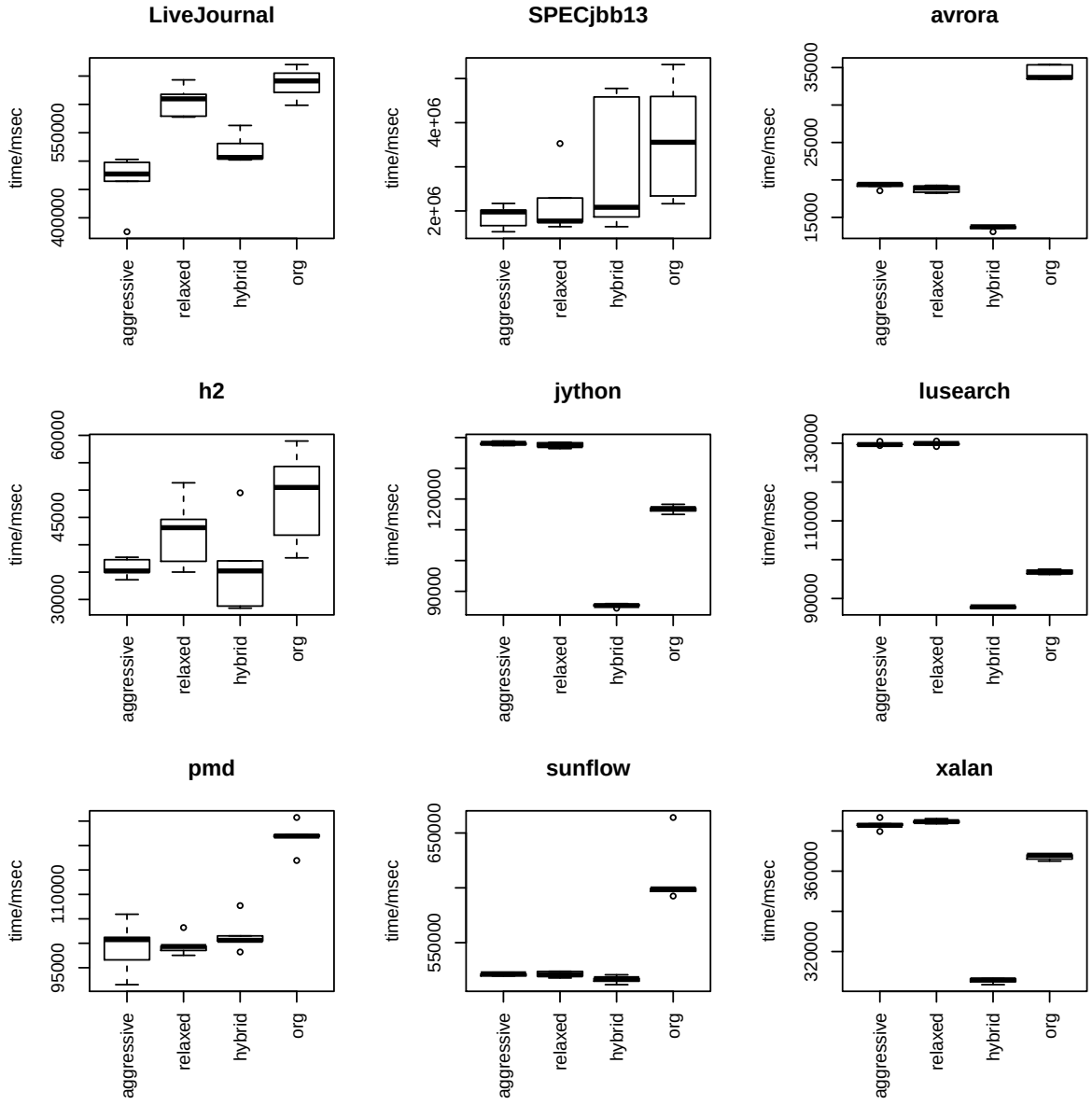


Figure 6.4: GC time (i.e. pause time) for our three optimization schemes. For small heaps (e.g. DaCapo programs), hybrid scheme gives the best results, whereas aggressive scheme is more effective for programs with larger heaps. (The default JVM is labelled Org.)

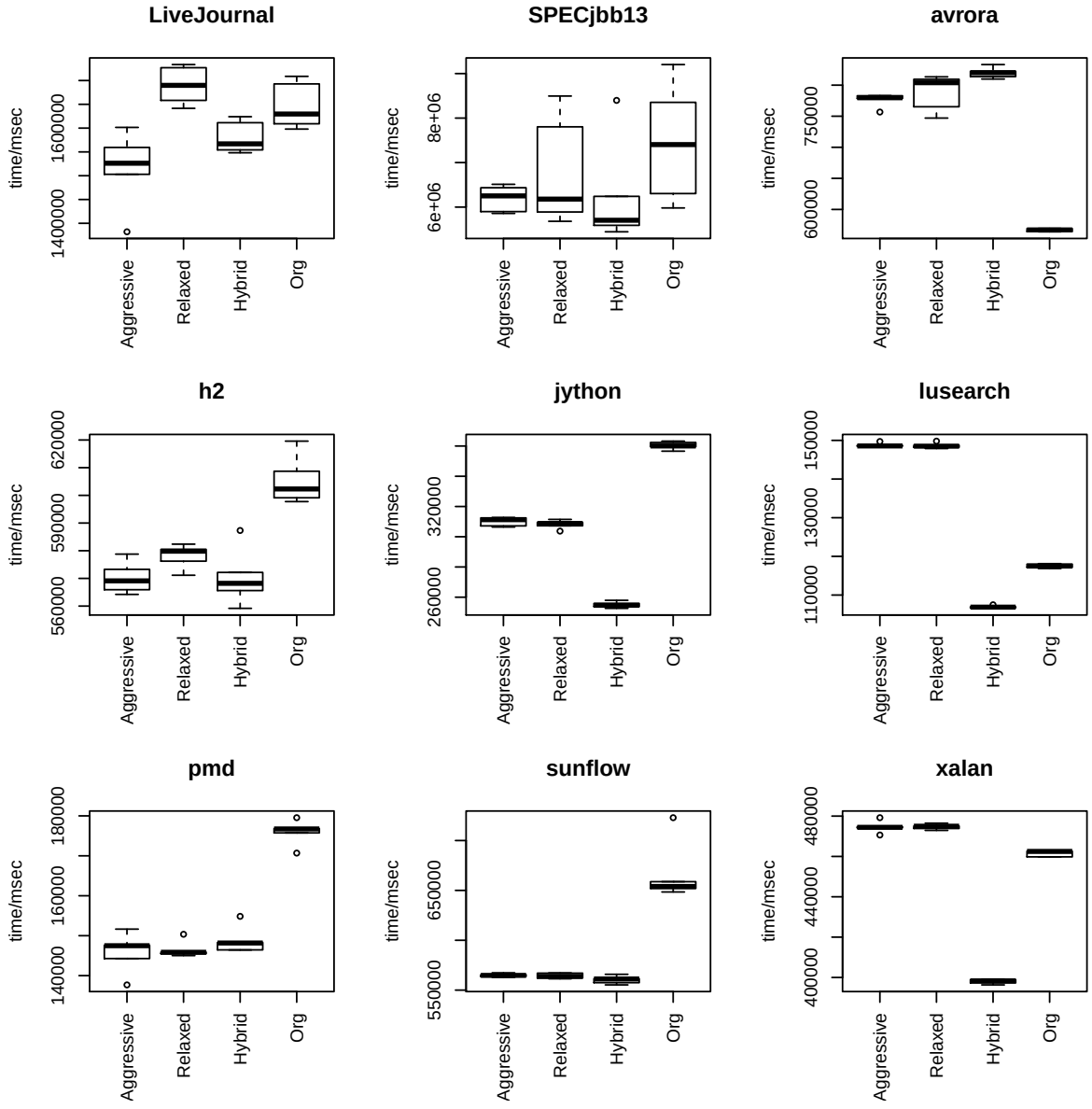


Figure 6.5: VM time (i.e. end-to-end execution time) for our three optimization schemes. At least one scheme provides better VM execution time than default (labelled Org) in most cases.

95% confidence interval of median. Proposed NUMA-aware parallel techniques for task distribution and work stealing outperform the default Hotspot Parallel Scavenge garbage collection policy (labelled *org*) in most cases. In addition, our optimization schemes incur less results variation due to the improved locality in rooted sub-graphs.

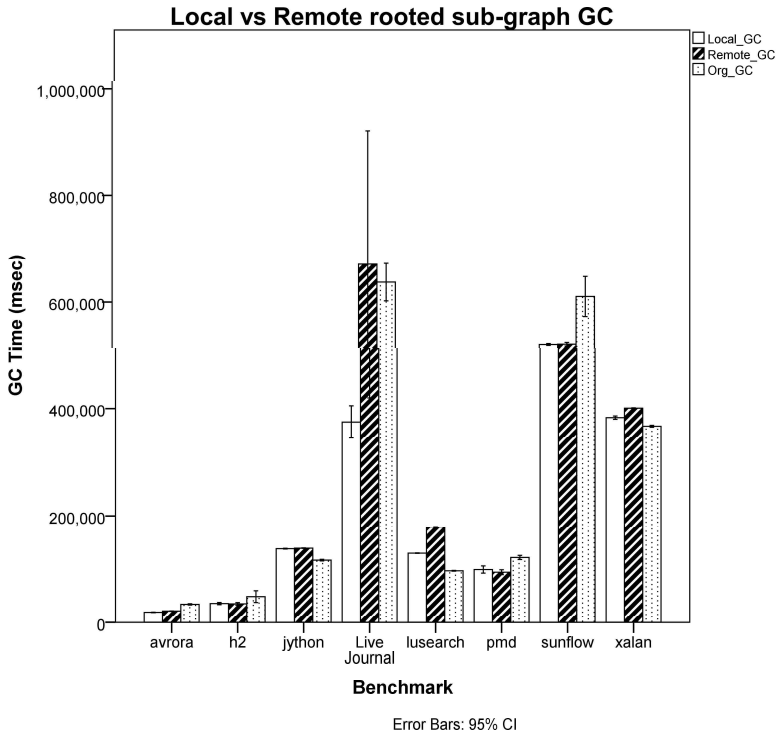
For workloads that require a small heap size, represented by the DaCapo benchmark suite, we can observe that *Hybrid scheme* is the best choice for high performance. The hybrid optimization scheme speeds up the garbage collection performance by up to 2.52x and never degrades it. This means that stealing work from remote NUMA nodes would balance the load over the collector threads or reduce congestion in the local NUMA node. In both cases, data locality cannot be the only optimization objective for NUMA systems. However, not all DaCapo benchmarks follow the same performance trend. DaCapo/h2, pmd and sunflow obtain relatively similar results to other optimization schemes. For aggressive and relaxed optimization schemes, garbage collection performance is better than the default JVM time in four DaCapo benchmarks: avrora, h2, pmd and sunflow.

We can notice that locality is vital to programs that have large heaps. Our optimization schemes improve Neo4j/LiveJournal garbage collection performance by 37%, 22%, and 5% for aggressive, hybrid, and relaxed schemes respectively. With the aggressive scheme, SPECjbb2013 obtains improvement in garbage collection performance by 91%.

In terms of VM execution time, the VM performance follows the garbage collection performance in all benchmarks except DaCapo/avrora. It uses too small heap size and exhibits a static memory layout. Avrora has shown negative performance results in previous research, for example, [Kalibera et al., 2012] [Sartor and Eeckhout, 2012].

Our optimization schemes have shown better performance than the default Hotspot JVM. It is important, though, to show that this performance gain is a result of improved NUMA local access not a caching effect. We test this for aggressive scheme and force garbage collection threads to process remote rooted sub-graphs. In this experiment, garbage collection threads of node *i* dequeue roots from NUMA queue *i+2*. Based on the rooted sub-graph hypothesis we expect that a garbage collection thread incurs 80% remote memory access.

Figure 6.6 depicts the pause time and total execution time results. The graph shows three bars: local access, which represents the aggressive scheme, remote access, and the default Hotspot pause times. The aggressive scheme has shown that it is the best optimization for large-heap benchmarks; therefore, remote access for Neo4j/LiveJournal is the worst result, which indicates that NUMA remote access has a major impact on garbage collection performance. DaCapo/xalan and lusearch results are similar to LiveJournal, whilst avrora, h2, jython, pmd, and sunflow show no impact. We have shown that the hybrid scheme is the best optimization for small-heap benchmarks.



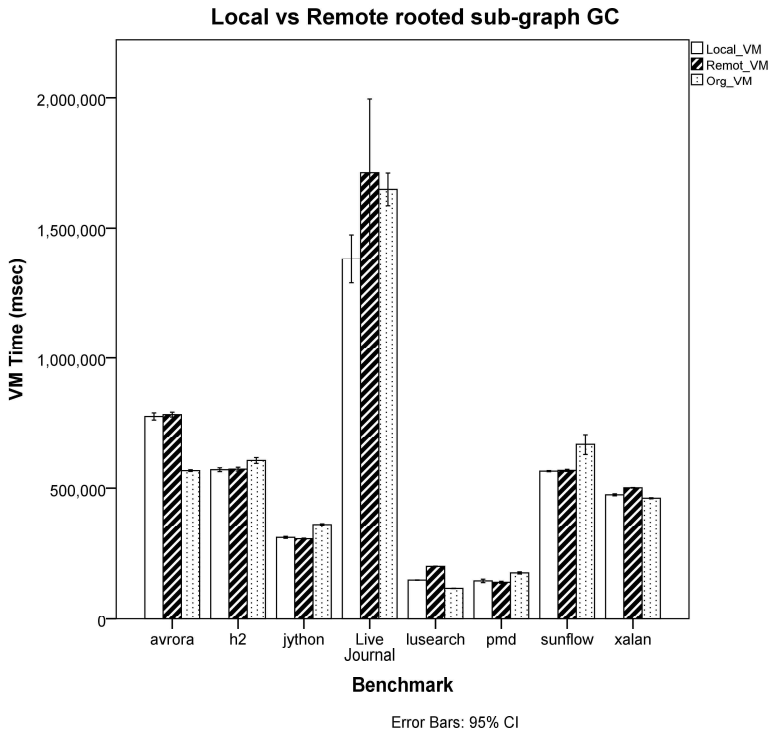


Figure 6.6: GC time and VM time comparison between local access, remote access, and the default JVM. GC and VM times for remote access is higher than local and the default JVM for LiveJournal benchmark.

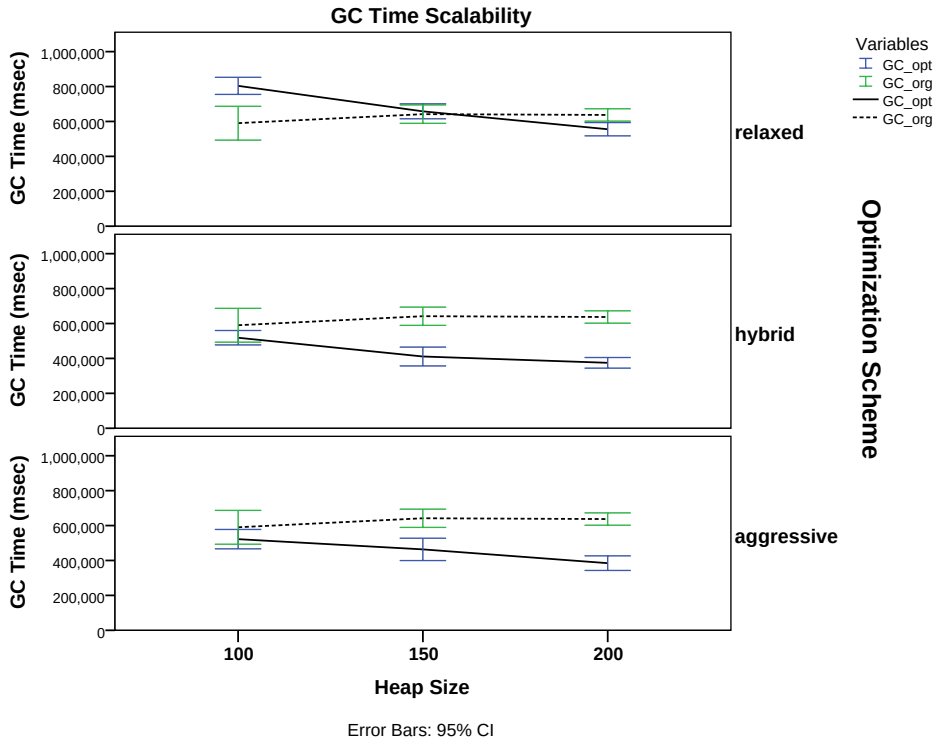


Figure 6.7: GC time scaling with heap size for Neo4j/LiveJournal. GC time decreases with heap size for our optimized versions, whereas the original implementation does not show any scaling.

6.4.4 Scalability

When applications require a large heap size, the root scanning task may consume a lot of time in scanning the old-to-young references. A large number of live objects were discovered through scanning the card table; hence, the card table scanning tasks account for the majority of garbage collection pause time. Our experience is that for heap sizes above 100GB, scanning the card table often takes hundreds of seconds.

In this section, we investigate the impact of increasing the heap size on our optimization schemes. We set the following heap sizes: 100, 150, and 200 GB to Neo4j/LiveJournal, the most memory-intensive workload in our benchmarks. Figure 6.7 and Figure 6.8 depicts the garbage collection pause time and the VM time results. Intuitively, as the heap size increases, the number of garbage collection cycles decreases. However, the original Hotspot garbage collection implementation shows a slight increase in the pause time. We argue that this increase is due to the time consumed by scanning and processing rooted sub-graphs in the card table. In particular, there are three aspects that could impact these results. First, old-to-young rooted sub-graphs tend to be deep and spend much processing time. Second, as discussed in Section 6.4.2, old-to-young roots attain poor locality between objects in their rooted sub-graphs. Therefore, the garbage collection thread incurs significant remote

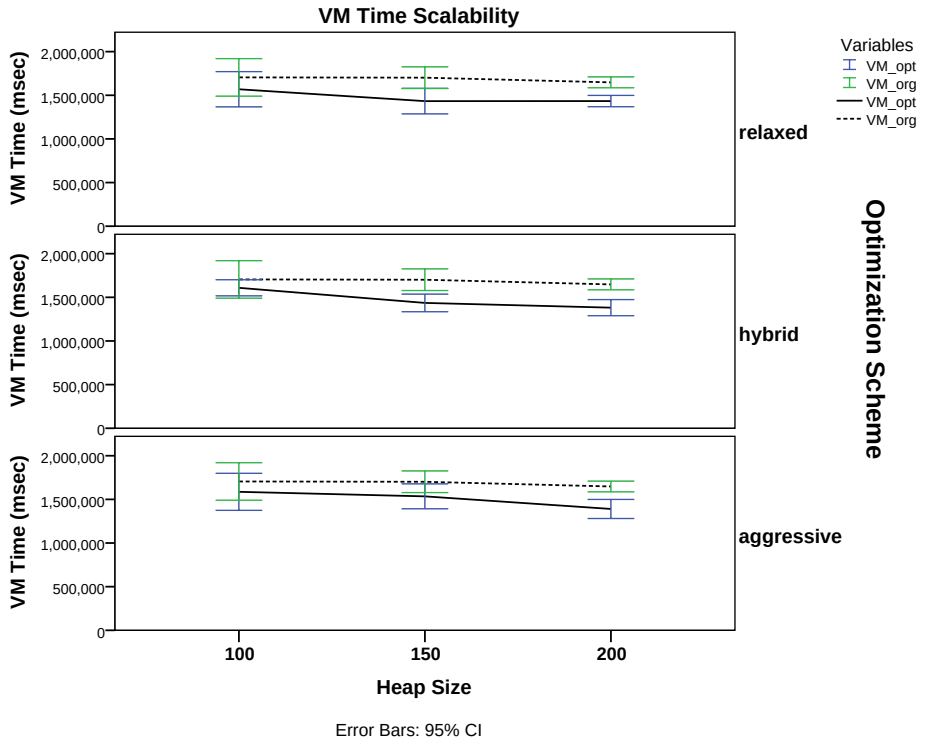


Figure 6.8: VM time scaling with heap size for Neo4j/LiveJournal. VM time decreases with increased heap size for our optimized versions, whereas the original implementation does not show any scaling.

access overhead. Third, deep rooted sub-graphs are likely to allow other garbage collection threads to steal some of their references; thus, stolen references would break connected object locality and disperse objects across NUMA nodes.

Our three optimization schemes improve the second and third aspects. By preserving the rooted sub-graph integrity and imposing NUMA awareness on work stealing we are successful in scaling the garbage collection substantially. In particular, relaxed scheme outperforms the original Hotspot garbage collection policy at 200GB heap size, by just processing rooted sub-graphs first before entering the work stealing phase. As a result, applications with a large heap size can benefit from the knowledge of NUMA topology to improve their memory access behavior.

For VM performance scalability, we can see that the original Hotspot garbage collection policy provides a steady VM time over the three heap sizes. Aggressive optimization scheme follows the garbage collection performance trend and obtains better scalability results. On the contrary, hybrid and relaxed optimization schemes observe better VM performance but only moderate scaling with increased heap size.

6.5 Related Work

Previous research proposes allocating related objects close to each other to improve locality. There are various criteria to choose which objects can be co-located. Chilimbi and Larus [1999] suggest considering temporal access patterns, whereas Shuf et al. [2002b] use data type for co-allocation. For graph-based workloads, graph traversal order can improve object locality. Wilson et al. [1991] introduce a hierarchical decomposition traversal order. This involves two different queues: small queues for sub-graphs such that objects reside in a memory page, and a large queue to link these small queues. In our NUMA-aware garbage collector, we use two queues: NUMA queues for root references and thread-local queues for rooted sub-graphs. Huang et al. [2004] attempt to co-locate frequently accessed objects in hot methods. While a program executes, they sample hot method fields and at garbage collection time, referents of hot fields are copied with their parents. Rather than sampling, our implementation relies on system calls to obtain object location at garbage collection time.

Thread-specific objects that are accessed by the owner thread only can be allocated locally, [Anderson, 2010, Jones and King, 2005, Domani et al., 2002, Marlow and Peyton Jones, 2011, Steensgaard, 2000]. Intuitively, newly allocated objects are initially stored in thread-local heaps until remote objects reference them. Consequently, referenced objects are promoted to a shared heap. Zhou and Demsky [2012] design a master/slave collector, where slave threads collect thread-local heaps only. If there is any reference to non-local objects, the slave thread sends a message to the master thread to route it to the target slave thread to mark it as live. In our algorithm, every garbage collection thread processes objects that reside locally in the same NUMA node.

The existing NUMA-aware collectors take into account the object location before and after garbage collection time. Tikir and Hollingsworth [2005] sample memory accesses during program execution, and then move objects at garbage collection time to the NUMA node of the thread accessing that object most frequently. Ogasawara [2009] identifies the dominant-thread of each live object, for instance the thread holding the object lock. He obtains the dominant thread for every live object and moves the object to the dominant-thread's NUMA node.

Connected objects in the object graph share various attributes. Hirzel et al. [2002] study several connectivity patterns that are related to the object lifetime. Results show that connected objects that can be reached only from the stack are shortlived; whereas, objects that are reachable from globals stay alive for a long time. Furthermore, objects that point to each other die at the same time. Object connectivity is strongly related to NUMA locality as well. In Chapter 5 we have shown that a high proportion of connected objects that descend from a root reside in the same memory node as the root.

Parallel garbage collection algorithms attempt to keep cores busy processing the reference graph. Conventional parallel techniques create a per-thread work list and populate the list with work tasks accessible by the owner thread. For load balancing, idle threads that run out of work, steal tasks from pending queues [Flood et al., 2001, Endo et al., 1997, Siebert, 2008, Wu and Li, 2007]. Nonetheless, such *processor-oriented* algorithms pay no attention to object locality; hence threads may incur overhead for accessing remote objects.

Memory-oriented parallel garbage collection algorithms take the memory location into account. These algorithms partition the heap into segments and assign a garbage collection thread to one or more segments. Threads process only local references and whenever they find references to remote objects, they push them into a queue of the corresponding segment [Chicha and Watt, 2006]. Alternatively, Shuf et al. [2002b] enqueue references to remote objects into a shared queue to enable other garbage collection threads to process them. For load balancing, queues are locked and garbage collection threads need to acquire the lock to dequeue live objects [Oancea et al., 2009]. These studies do not map memory regions to NUMA nodes. A memory segment boundary might span multiple physical memory frames. Further, a garbage collection thread may process remote memory regions. Improvement could be possible by matching memory regions to NUMA nodes.

To balance the load, this means that a garbage collection thread needs to separate child objects from their parents and this can negatively affect object locality. Gidra et al. [2011] observe that disabling load balancing could improve program performance for some applications. Muddukrishna et al. [2013] propose a locality-aware work stealing algorithm, which prioritizes the work eligible for stealing according to the distance between NUMA nodes in a system with multi-hop memory hierarchy. In this algorithm, an idle thread on a node attempts to steal work from the 'nearest' pending queues. Olivier et al. [2011] propose a hierarchical work stealing algorithm. They devote one third of the running threads to steal work and push stolen work into a shared queue for local threads. Our aggressive optimization scheme allows garbage collection threads to steal work only from NUMA-local pending queues to improve object locality.

6.6 Conclusion

We have shown that a NUMA-aware copying collector based on per-NUMA node task distribution is able to preserve much of the rooted sub-graph locality that is inherent in mutator allocation patterns. Using NUMA-local root reference as locality heuristic, except old-to-young roots, enables garbage collection threads to process local references and steal references from sibling threads. Although locality improvement is the main objective of this study, some applications gain performance when processing remote rooted sub-graphs, due

to the imbalance allocation between NUMA nodes. Therefore, we study three optimization schemes: aggressive (process only local roots), hybrid (process any root, but with preference to local roots, and steal only from NUMA-local queues), and relaxed (process any root and steal from any queue).

The study results show that for large-heap applications, aggressive scheme performs better than hybrid and relaxed schemes. This result is due to the fact that NUMA-local queues contain a large number of roots and garbage collection threads process mostly local references, which reduce the cost of remote memory accesses. Adversely, small-heap applications benefit much from hybrid scheme. In this scheme, some garbage collection threads run out of work (root processing) and attempt to steal references from sibling threads only. This behavior suggests that, for our small-heap applications, memory allocation is imbalanced between NUMA nodes. Our optimization schemes have shown significant benefits—with improvements in garbage collection performance ,on average 13% for aggressive scheme, 23% for hybrid scheme, and 8% for relaxed scheme.

This study focused on stop-the-world garbage collector, which mutator threads halt to enumerate root references. For concurrent garbage collectors, our optimization schemes may face challenges because the root enumeration phase is based on individual threads. This issue and other issues such as other runtime systems will be discussed in Section 8.3.1.

We believe that there are further possible improvements based on not only preserving locality of reference sub-graphs in single NUMA nodes, but also using NUMA-local collector threads to operate on these rooted sub-graphs. In this study, we rely on expensive system calls to identify NUMA-local tasks for collector threads—but cheaper techniques are presented in recent literature [Gidra et al., 2015].

In summary, garbage collection implementations should be able to preserve intra-node reference graph locality as much as possible in order to enable subsequent low-latency access times for both mutator and collector threads.

CHAPTER

7

NUMA-AWARE GARBAGE COLLECTION THREAD MANAGEMENT

Contemporary multicore processors provide abundant parallel compute resources to increase application performance. Prevalent server-class NUMA machines are shipped with diverse hardware configurations that parallel programs should take into account to efficiently utilize the system. One parallel programming consideration is to decide how many hardware threads the application should use in order to execute the workload. This decision is a non-trivial problem. Many multi-threaded applications set the number of executor threads equal to the number of cores available in the system. Nonetheless, allocating full resources in the system to the application may result in a suboptimal performance.

The Hotspot JVM addresses this problem by using an adaptive garbage collection thread policy. This policy changes the number of collector threads at each collection cycle based on various factors. However, this policy is shown to work ineffectively in modern machines because it is generally designed for specific legacy platforms. This chapter investigates the performance of the existing adaptive collector thread policy of the Hotspot JVM when running on NUMA machines. First, we study this policy and show how and why it is not suitable for modern NUMA machines. Second, we alter the number of garbage collection threads and measure the garbage collection throughput for minor and major collections, i.e.

the copying and the mark-compact collectors. Third, we use the throughput results to design and implement static and dynamic optimizations to estimate the optimal number of collector threads.

The contributions of this chapter are three-fold:

1. It quantitatively studies the garbage collection performance behavior when changing the number of collector threads for minor and major collection. The study involves a set of hardware performance counters that AMD Opteron processors support. In our machine, there are four hardware performance counters, i.e. one for each socket. We use these counters to measure off-node traffic between NUMA nodes. In addition, we measure the application execution and pause times.

2. It estimates the optimum number of collector threads by fitting the garbage collection's throughput results of each workload to quadratic curves. Based on the results, we implement a static optimization that sets and fixes obtained optimal number of collector threads throughout program execution.

3. It implements a runtime search-based optimization to dynamically predict the optimal number of collector threads for each collection cycle. This optimization is based on a *gradient-ascent* search algorithm, which predicts the appropriate number of threads prior to each collection. The results show an average of 25% and 5% improvements to the garbage collection performance for DaCapo and overall benchmarks, respectively.

The organization of this chapter is as follows: Introduction and motivation are presented in Section 7.1. The Hotspot default policy for adapting the number of garbage collection threads is described and discussed in Section 7.2. Section 7.3 empirically studies the impact of scheduling different numbers of collector threads on collection throughput. Section 7.4 designs and implements a static optimization for selecting the optimal number of collector threads, whereas Section 7.5 provides a runtime policy for adaptive garbage collection threads. Section 7.6 compares and contrasts this work with closely related research, and Section 7.7 summarizes this chapter.

7.1 Introduction

Multicore systems allow software developers to attempt to improve application performance by running the workload on parallel hardware. To parallelize programs, developers have designed several parallel programming models to manage and efficiently utilize the parallel hardware. As a result, programs are executed with a high number of threads. In the context

of NUMA garbage collection, performance may not be improved by running a high number of garbage collection threads; *this will be explained in Section 7.3*.

A great deal of research has focused on improving data locality. Chapter 2 and Chapter 5 provide various examples for data locality improvements for NUMA machines. Furthermore, hardware manufacturers have increased interconnection link speeds between NUMA nodes, hence, memory access latency is reduced. Nevertheless, other NUMA issues, for example, bandwidth saturation caused by the imbalanced memory allocation and congestion on memory controllers remain under research. Gaud et al. [2015] argue that NUMA congestion at memory controllers and buses is more serious than data locality.

A large body of research on NUMA optimization attempts to improve application threads performance, in general. However, garbage collection for NUMA heaps faces similar challenges. Whilst co-locating data in a NUMA node may improve data locality, congestion can occur in the NUMA node's local memory system. In Chapter 6, we attempted to improve data locality for garbage collection with three schemes. The aggressive scheme limits garbage collection threads to process rooted sub-graphs in local nodes only. However, results show that imposing strict data locality may not be appropriate for NUMA systems. Adversely, the hybrid scheme outperforms the aggressive scheme because it enables utilizing remote resources.

In this chapter, we tackle NUMA off-node traffic from a different perspective: managing the number of collector threads. Common practice is to fully utilize the system resources to execute the workload. However, anecdotal discussions reveal that scheduling many collector threads could degrade application's performance [Printezis, 2009]. The garbage collector encounters parallelism issues, for example lock contention. Gidra et al. [2013], Iyengar et al. [2012] identify and improve some of these issues .

Previous research suggest several proposals to mitigate resource contention such as balancing memory allocations between NUMA nodes [Dashti et al., 2013]. These studies attempt to solve problems with an assumption that full system's resources are being used. In contrast, Hotspot JVM uses an adaptive policy for garbage collection thread management. This policy is implemented in method `calc_default_active_workers` of the Hotspot source code [Hotspot_Source_Code]. This policy emerged in response to performance degradation when utilizing full resources [Printezis, 2009]. Although it adapts the number of collector threads at each collection cycle, the policy enforces an upper limit on the number of threads. Because the policy was designed for specific set of machines (SUN systems from the 2000s) and it has not been updated yet, the policy generates a fixed number of threads on modern multicore platforms. Therefore, using the system's full resources or relying on the existing Hotspot policy may not yield an optimum garbage collection performance.

Choosing an optimal number of parallel threads to execute a garbage collection workload is

a non-trivial problem. In this chapter, we empirically study the impact of selecting different numbers of collector threads on garbage collection's throughput and NUMA off-node traffic. To the best of my knowledge, the impact of choosing the number of collector threads has not been studied in the context of NUMA architecture. Results show that the mean collection throughput peaks at a threshold number of collector threads, and then degrades as we add further threads to the collector thread pool.

In further analysis, we argue that not all garbage collection phases require the same number of threads. Results show that the young generation collection's work is, in general, best carried by fewer threads compared to the full heap collection. We employ curve fitting for collection throughput to estimate the optimal number of collector threads. In addition, we use a runtime search-based optimization, called a *gradient-ascent* [Snyman, 2005] to estimate optima for minor and major collections separately. At every garbage collection cycle, the gradient-ascent technique learns from previous collection's throughput and predicts the number of threads needed for the current collection. On average, proposed optimizations improve collection throughput over the default policy by an average of 21% for DaCapo and 5% for all benchmarks.

7.2 Hotspot GC Threads Management

The Hotspot JVM manages the number of collector threads in several ways. Firstly, it provides a command-line flag to allow the user to set the number of parallel collector threads, Oracle [2016]. A user's explicit thread configuration at VM initialization overrides all other configurations and it is used and remains *constant* throughout every garbage collection cycle, for both minor and major collections.

Secondly, if the user does not change the number of collector threads explicitly, then the HotSpot activates its *adaptive* garbage collection thread management. This policy adapts the number of collector threads, if required, at every collection cycle. As a common practice, Hotspot developers do not engage full system resources to the garbage collection. Anecdotally, consuming full resources yields suboptimal performance [Printezis, 2009]. Therefore, the Hotspot JVM sets an upper limit for the number of collection threads M using the following equation:

$$M = 8 + \big((P - 8) * \frac{5}{8}\big) \tag{7.1}$$

where P is the number of processors. For example, if a system has 32 cores then the maximum number of garbage collection threads is 23. If the number of cores is equal to or less than 8, then the policy uses all cores.

The adaptive policy manages the number of collector threads at every collection cycle using

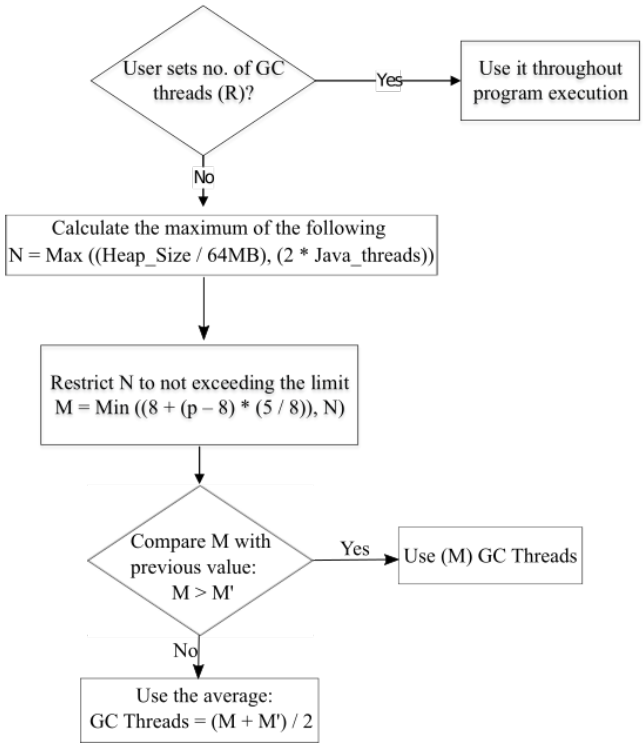


Figure 7.1: Hotspot JVM policy for setting a dynamic number of garbage collection threads.

two key factors. First, there is a heuristic that the number of collector threads should be at least double the number of mutator threads, T_{mutator} [Hotspot_Source_Code]. This is based on the assumption that the amount of memory used by a single Java thread would be best collected by two garbage collection threads. Second, the number of garbage collector threads should be related to the heap size, i.e. large heaps are collected by a large number of threads. Hotspot assigns a garbage collection thread for every 64MB of the heap. Consequently, the number of collector threads for nth collection cycle T^n_{GC} is calculated as:

$$T^n_{\text{GC}} = \max\big(2T_{\text{mutator}}, \frac{\text{heapsize}}{64\text{MB}}\big) \tag{7.2}$$

If the number of collector threads exceeds the upper limit M, then only M threads will be used at nth collection cycle. At every collection cycle, the number of collector threads at the previous collection is taken into account. If the calculated number of collector threads for the nth garbage collection is bigger than the number of threads for $(n-1)^{\text{th}}$ garbage collection, then T^n_{GC} is applied. Otherwise, the new number of collector threads is calculated as:

$$T^n_{\text{GC}} = \frac{\max\big(2T_{\text{mutator}}, \frac{\text{heapsize}}{64MB}\big) + T^{n-1}_{\text{GC}}}{2} \tag{7.3}$$

This means, the rate of change in collector thread count is asymmetric. When the runtime decides to increase the number of threads, it does so instantly. However, decreasing the number of GC threads is done gradually. This asymmetric change is eager to keep the number

of garbage collection threads high (around the upper limit) to take advantage of the parallel resources. Figure 7.1 shows the Hotspot adaptive thread policy schematically.

7.3 Impact of Varying the Number of Collector Threads on Throughput

Hotspot implements a throughput-oriented garbage collection policy, called **Parallel Scavenge**. This collector aims to provide high collection throughput by using parallelism, adaptive heap sizing policy, adaptive garbage collector thread management policy, and a service level agreement (SLA). With modern NUMA systems, the adaptive garbage collector threads management policy is shown as not to work efficiently and it gives a fixed number of threads throughout program execution. Along with the theoretical analysis discussed in Section 7.2, this section studies the impact of choosing different number of collector threads on collection's throughput.

The first objective of this study is to observe collection's throughput behavior when specifying a different number of threads at VM initialization each time. Note that we measure collection throughput for minor and major collections separately because they have different characteristics. The minor collector is a copying collector, whereas the major collector is mark-compact collector. Throughput is calculated as size of collected heap space divided by wall clock time of garbage collection. The number of collector threads for each workload is fixed and it ranges from 2 to 32. For each (workload, collector threads) combination, we take 20 measurements and report arithmetic mean. All error bars on graphs show 95% confidence interval.

The second objective is to measure off-node NUMA traffic using hardware performance counters. This dissertation considers using hardware performance counters as a *novel* contribution to the research. Off-node memory events provide an indication about whether increasing the number of collector threads can causes NUMA congestion on the system's interconnection links. It also can show whether or not data allocation is balanced between NUMA nodes. Although our AMD system is an eight-node machine, it supports four hardware performance counters only to measure off-node events. This limitation forces us to emulate a machine of four nodes. Accordingly, we configure the machine to be of four nodes, 32-cores, and 250 GB memory using *numactl* Linux NUMA tool. We use the Likwid [Treibig et al., 2010] profiling tool to count memory events, i.e. read/write operations, that pass through off-node interconnection links. These events include memory operations from both mutator and garbage collector threads. The experiments ran for 20 times and the results show the sum of all links averaged by the number of cores.

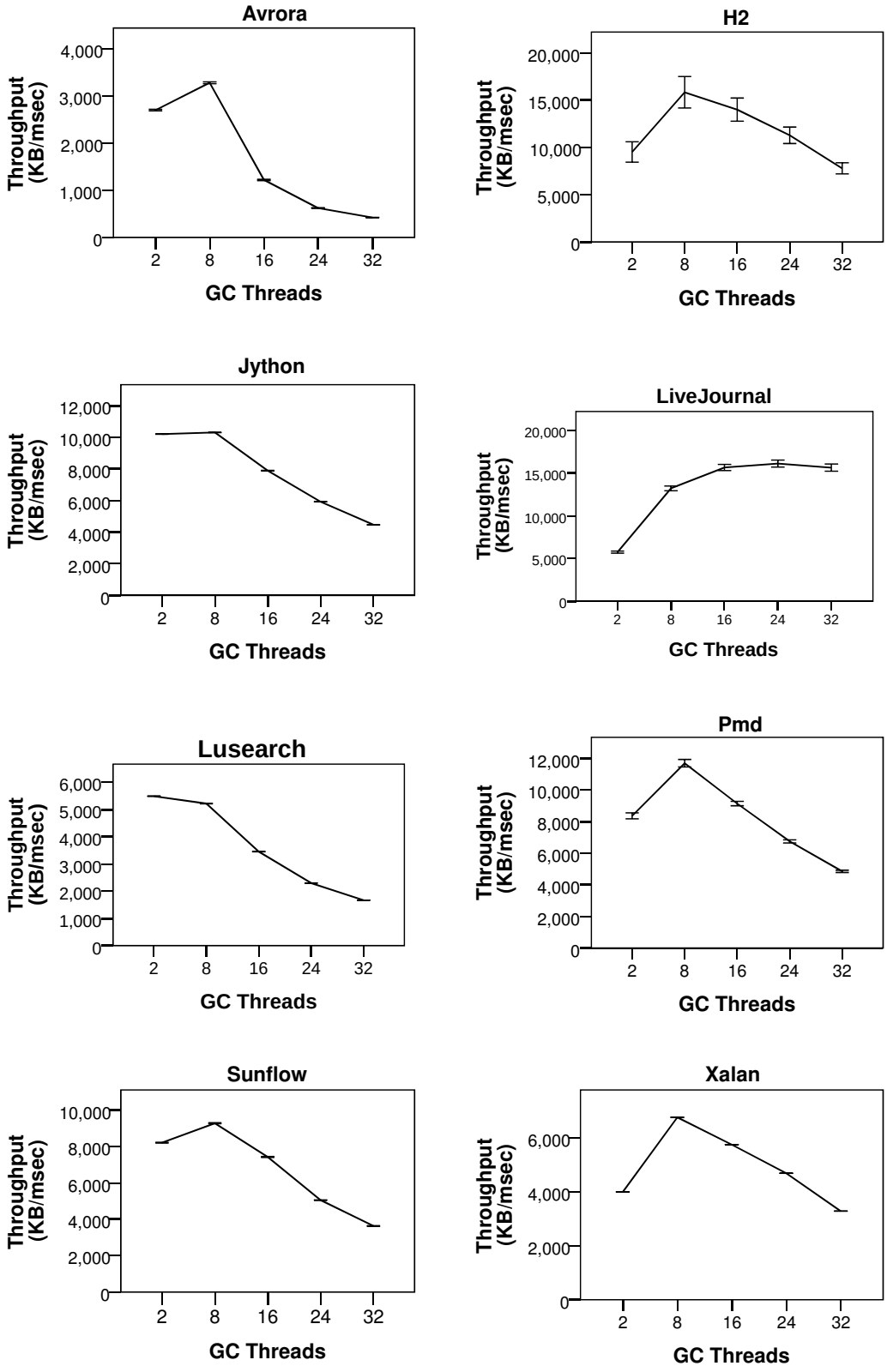


Figure 7.2: Minor collection throughput varies with number of collector threads (higher is better)

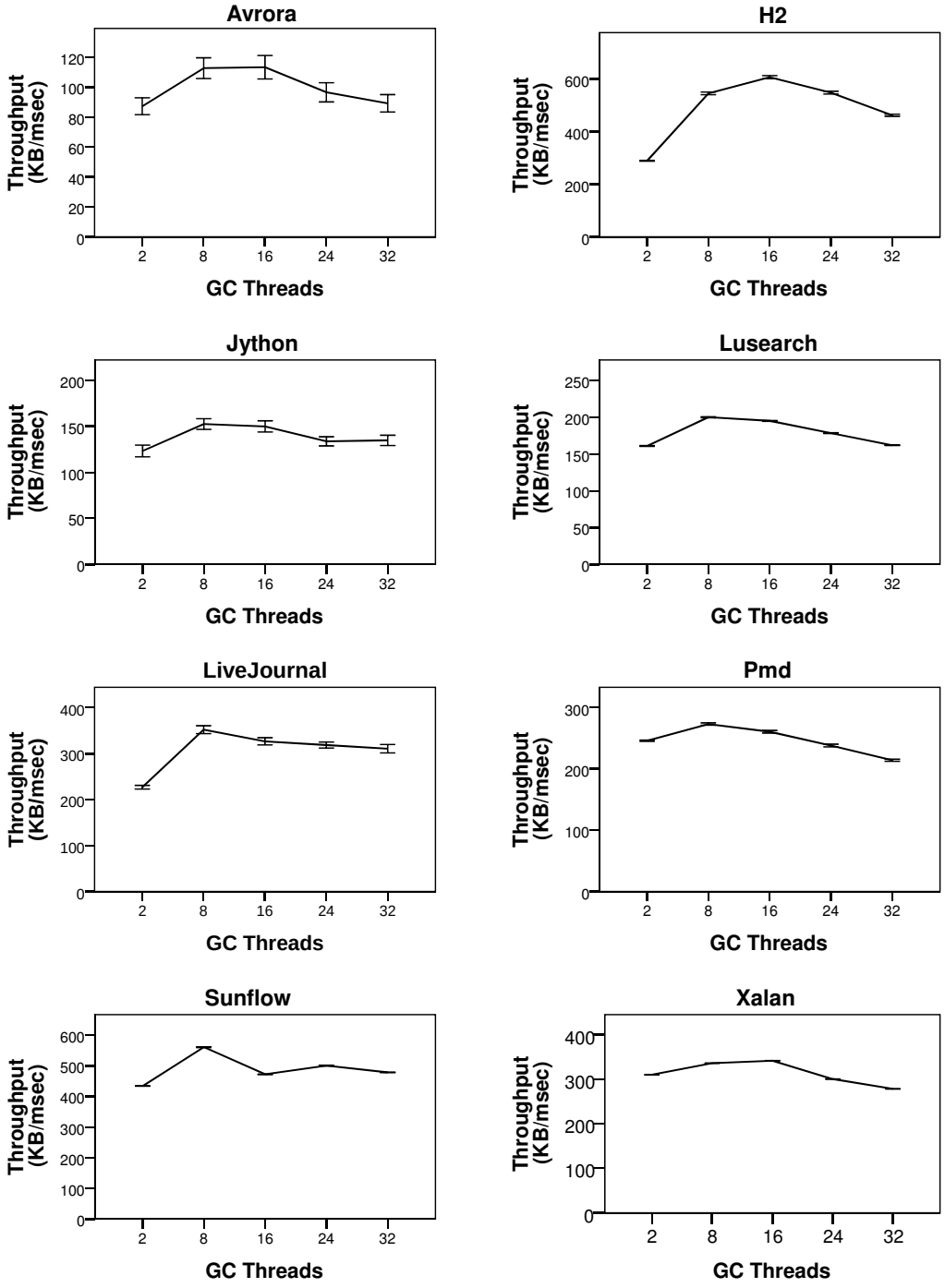


Figure 7.3: Major collection throughput varies with number of collector threads (higher is better)

The results in Figure 7.2 show that minor collections throughput improves as we increase the number of collector threads until it reaches a maximum value, then performance degrades beyond this threshold. For avrora, h2, pmd, xalan, and sunflow graphs, the threshold number of collector threads is around 8 threads; whereas livejournal continues to improve throughput up to around 24 threads. Surprisingly, jython and lusearch show no advantage in having more than two threads.

Minor collections use a copying algorithm, where each collector thread has its own buffer in the target space and it is just bumping the pointer to copy objects. Therefore, the copying collector encounters less locking contention to allocate memory. On the contrary, copying objects to remote NUMA nodes leads to creating NUMA traffic [Gidra et al., 2011]; hence, smaller thread counts are more appropriate for minor collections. Livejournal exhibits a huge workload and its data lives long; thus, it may require more threads to carry out the collection work. In summary, minor collections prefer to schedule fewer threads, though for specific applications, with long-lived data, more collector threads are needed.

Figure 7.3 depicts major collection's throughput. It is clear that the threshold value for major collections is different from minor collections. For avrora, h2, jython, and xalan, the optimal number of threads is around 16 threads; whereas for lusearch, sunflow, pmd, and livejournal it is around 8 threads.

Major collections use a mark-compact algorithms, and the old generation size is generally much larger than the young generation. In addition, objects in the old generation are long-lived; thus, the number of collector threads is likely to be higher than the minor collections. We can see from the graphs that, in general, throughput variance is more narrow as we increase the number of threads compared to minor collection's throughput. Although throughput peaks are evident, other environmental or architectural issues such as efficient energy consumption or virtualization may benefit from fewer number of threads since scheduling additional collector threads would not make much difference.

The second objective of this study is to investigate the impact of running a high number of collector threads on NUMA off-node traffic. Figure 7.4 shows the absolute values for hardware performance counter averaged over the number of cores. Due to performance tool limitations, we are unable to measure memory events occur during the garbage collection time only, thus the curve shows off-node traffic for complete execution of each benchmark. The curve represents *the sum* of memory events pass through interconnection links. We would expect that as we increase the number of garbage collection threads, benchmarks issue high NUMA-traffic because threads would be mapped to multiple NUMA nodes and they are likely to cross NUMA nodes to access memory. However, not all benchmarks exhibit the same behavior. In jython, pmd, and livejournal graphs, 16 collector threads encounter low NUMA traffic. If we compare these results with Figure 7.3, Whilst avrora and h2 show

existence of traffic minima, lusearch, sunflow, and xalan exhibit rapid traffic growth off the nodes as we increase the number of collector threads. Since NUMA traffic measurements include mutator and garbage collection execution, analysis for the effect of increased number of garbage collection threads with the limitations in the existing profiling tools to the NUMA traffic is hard.

To sum up, using too many collector threads can degrade performance on NUMA machines. Furthermore, we observe that the optimal number of collector threads is *different* for minor and major collections, and for different benchmarks. Based on these conclusions, we can take advantage from this study to statically analyze the collection's throughput behavior and compute the optimal number of threads for each benchmark and for minor and major collections. The next section implements a static optimization for the number of collector threads.

7.4 Static Optimization

The existing adaptive garbage collection thread management policy produces a fixed number of threads for both minor and major collections. In addition, users can explicitly set the number of collector threads via a command-line switch for both collections. However, we have shown in the previous section that both options yield suboptimal garbage collection performance.

This section investigates the possibility of performing profile-based analysis on collection throughput to generate separate optimal numbers of collector threads for minor and major collections. Based on the observed throughput's behavior in the previous section, we can model the throughput's behavior by fitting it into a curve. The shape of collection throughput in Figure 7.2 and Figure 7.3 suggest that there is a maximum throughput value within the range of garbage collection threads we analyse. Therefore, a second degree polynomial function would be able to model this turning point. We will use the quadratic function $y = ax^2 + bx + c$ where y is garbage collection throughput and x is number of collector threads. We use SPSS version 21 to fit throughput data and estimate a quadratic curve for each benchmark.

The quadratic curve has a parabola shape. Based on our observed throughput behavior, we would say that, in general, throughput is enhanced as we increase the number of collector threads until a turning point after which throughput is worsened. This turning point indicates optimal number of collector threads we should use.

Figure 7.5 and Figure 7.6 show the fitted quadratic curves for the Dacapo and LiveJournal benchmarks for minor and major collections respectively. For minor collection graphs, we see wide parabolas for avrora, xalan, pmd, h2, and livejournal benchmarks. In contrast, for

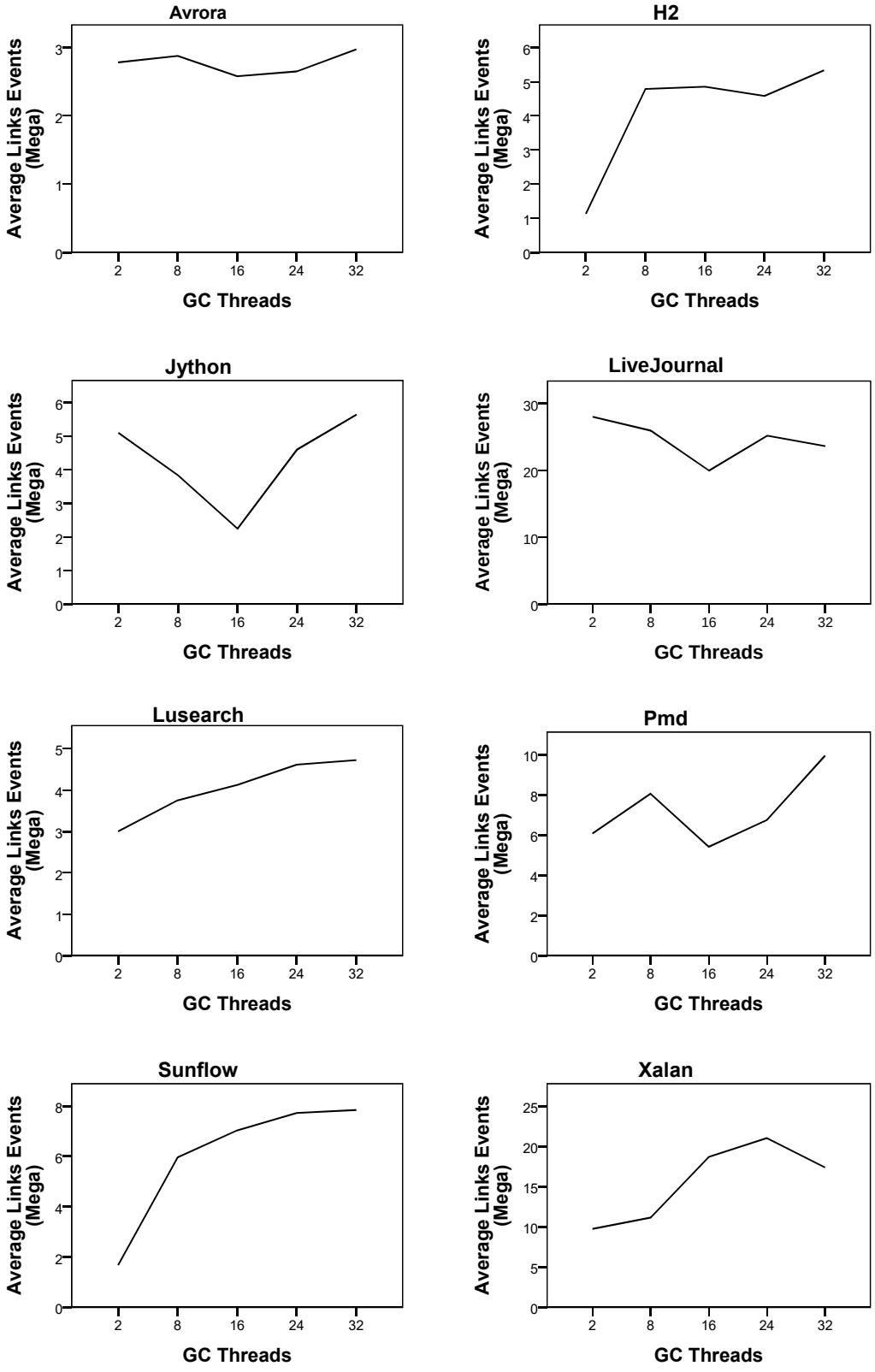


Figure 7.4: *Off-node memory events* over interconnect links between nodes; each line represents per-node average memory events over all four links (lower is better).

Application	minor threads	major threads
Avrora	9	16
H2	14	19
Jython	2	16
Lusearch	2	8
Pmd	9	10
Sunflow	2	17
Xalan	14	13
LiveJournal	23	19

Table 7.1: Optimum number of collector threads for minor and major collections

jython, sunflow, and lusearch, the initial value on the x axis is much better than other points on the curve, i.e. *minimum* collector threads yields better throughput. For major collections, parabolas shapes have a clear turning point indicating optimal number of threads.

SPSS computes the parameters for the quadratic formula. We can differentiate the quadratic formula to compute the location of the turning point on the parabola. Note that a single workload will have two optimum values—one for minor and one for major collections. Table 7.1 gives the computed optimum values for the number of minor and major collector threads for each benchmark.

Computed values for minor collections for avrora, jython, lusearch, pmd, and sunflow suggest to use few collector threads (≤ 9). In fact, jython, lusearch, and sunflow values are less than two but we set two threads as the lower bound for parallel threads. The throughput curve starts at high values then slides down as we increase the number of threads. Other programs like xalan, h2, and liveJournal exhibit more symmetry in the shape of the parabola. The number of collector threads for xalan and h2 benchmarks is 14, whereas liveJournal uses 23 threads.

For major collections, curves clearly indicate sweet-spots, where collection throughput peaks. The mark-compact collector requires a high number of threads for full heap collection. The optimal number of threads ranges from 8 to 19 across the benchmarks.

We implement static optimization by patching the JVM to allow different fixed thread settings for minor and major collections. Benchmarks ran for twenty times and we report arithmetic mean of pause times and VM times. Then we compare the computed static optimization results against default Hotspot adaptive policy. Figure 7.7 shows the garbage collection time for various workloads with these policies. For DaCapo benchmarks, the static optimization for garbage collection is significantly better (25% on average) than the Hotspot default adaptive policy. However, static optimization for the number of garbage collection threads of LiveJournal benchmark shows no improvement to the garbage collection performance with high pause time variation. By looking at the quadratic function of LiveJournal, the estimated

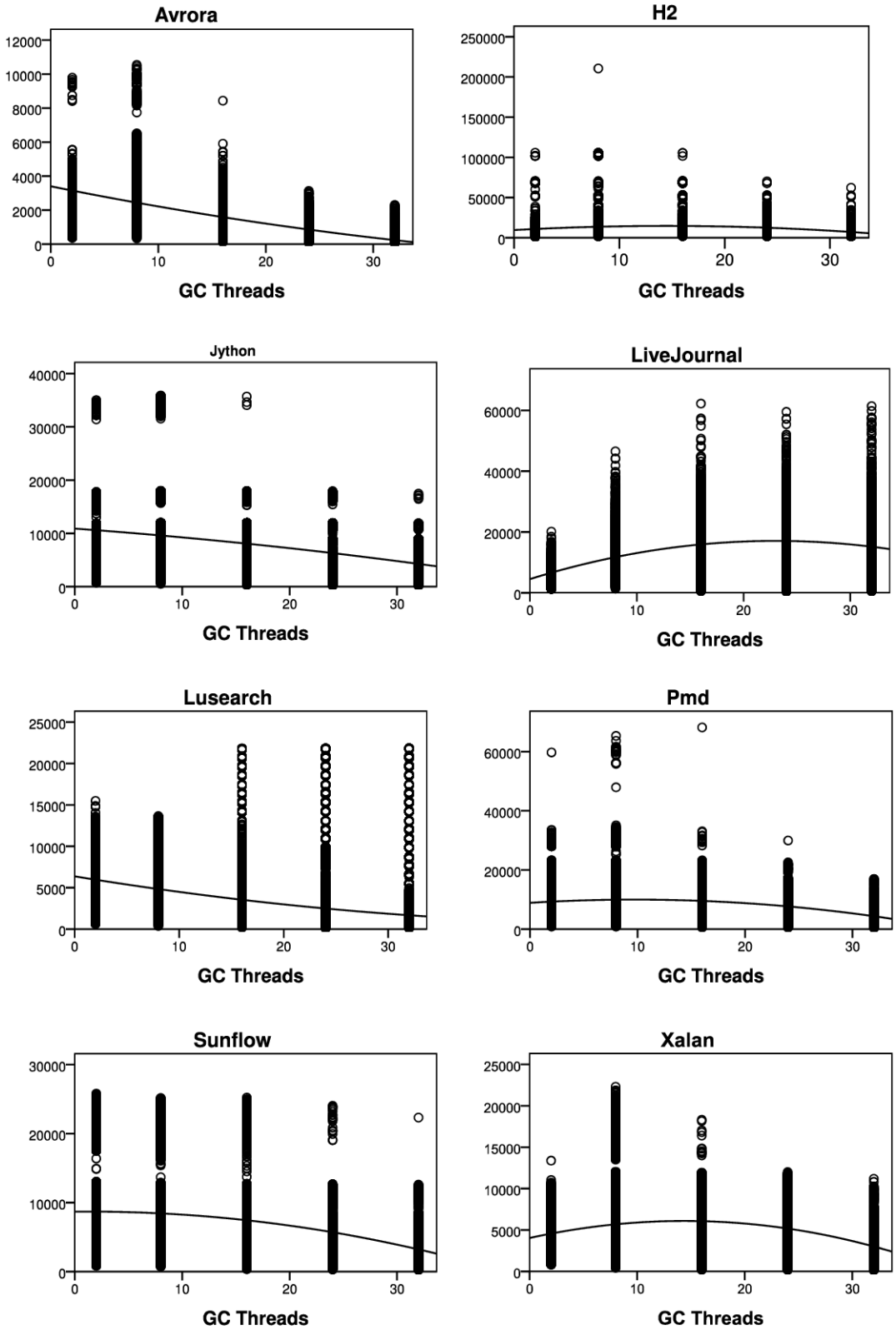


Figure 7.5: Fitted quadratic curves for benchmark GC throughput observations for minor collections

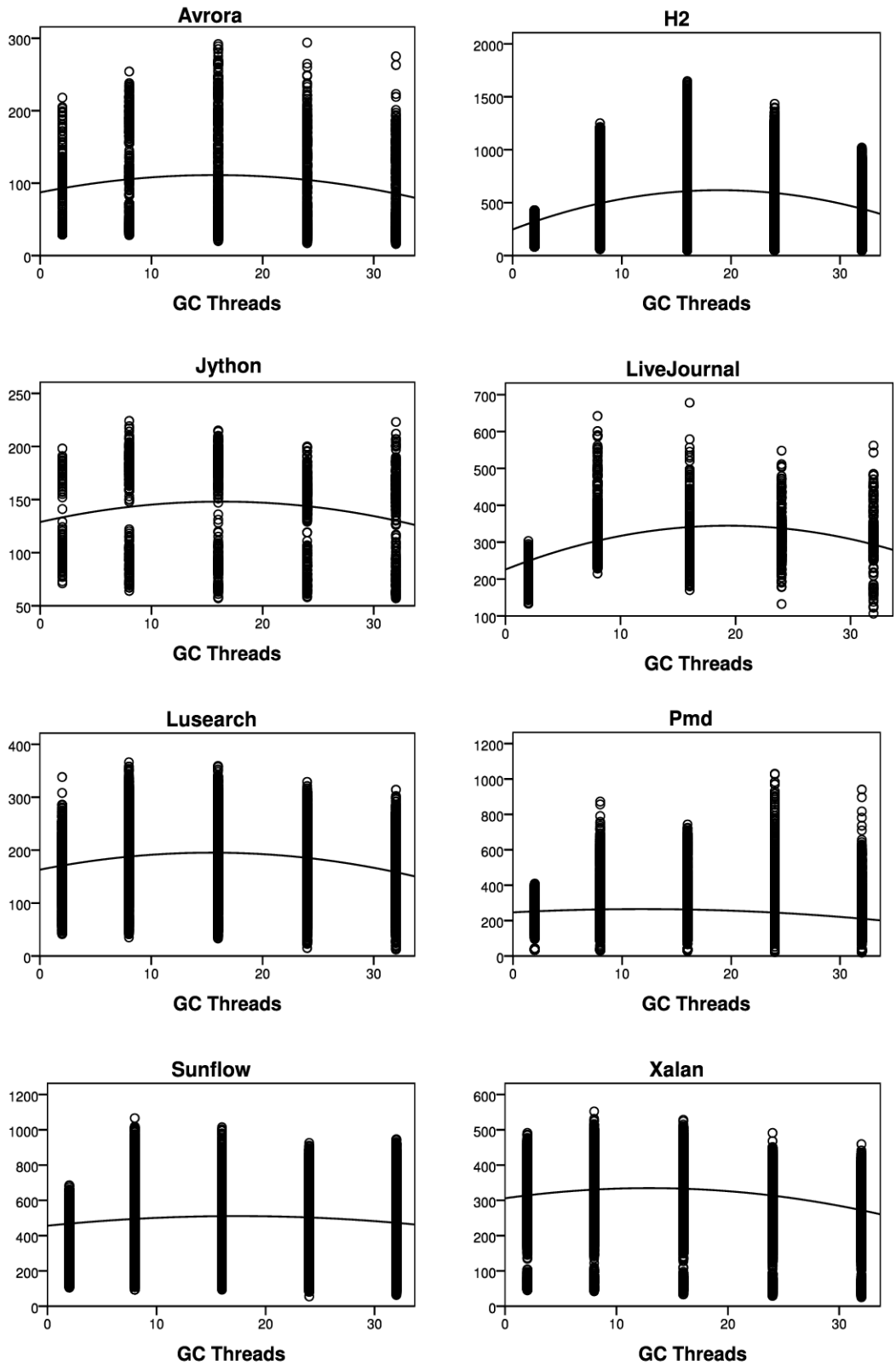


Figure 7.6: Fitted quadratic curves for benchmark garbage collection throughput observations for major collection

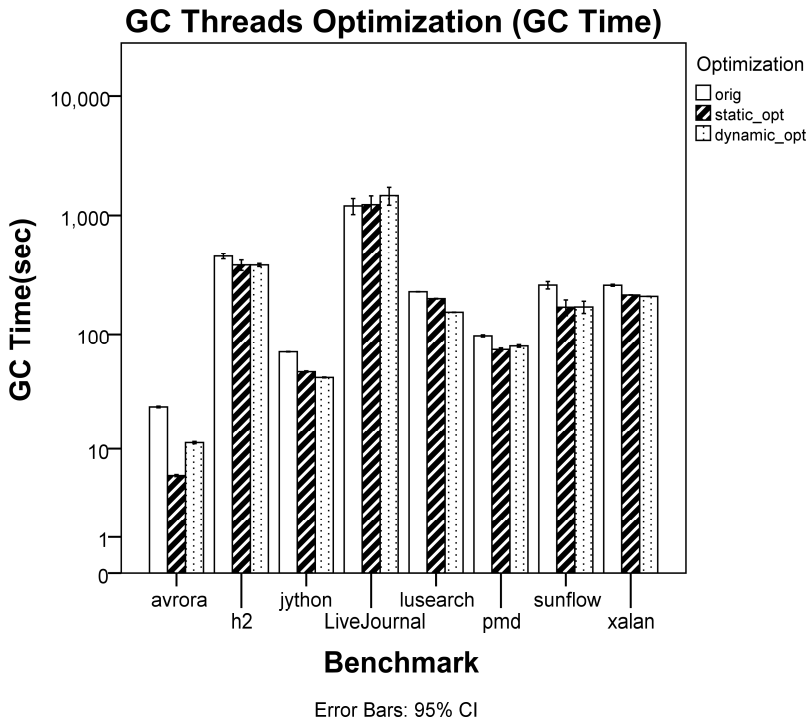


Figure 7.7: Garbage collection pause time comparison for various garbage collection threading policies (lower is better)

optima for major collection is 19, whereas its throughput graph (7.3) indicates that optima is 8. LiveJournal's major collection throughput shows no significant variation after 8 threads and the static optimization indicates that 23 threads (from the default JVM) and 19 threads (from the curve fitting) are almost the same.

Figure 7.8 shows the overall application execution time for the same workloads. In the majority of cases, the static optimization configuration leads to significantly better execution times than the Hotspot default adaptive policy. The total execution time for LiveJournal benchmark follows the same trend of pause time results, whereas avrora VM time is much worse than the default JVM, which is the same conclusion from Figure 6.5 page 99.

In summary, static optimal garbage collection thread management configuration significantly improves collection performance over the Hotspot default settings.

7.5 Dynamic Optimization

In the previous section, we set the number of collector threads explicitly on the command-line for minor and major collections and they remain constant throughout program execution. The optimum number of garbage collection threads may vary during execution, e.g. if a program goes through phase changes, or if several distinct programs are chained together in

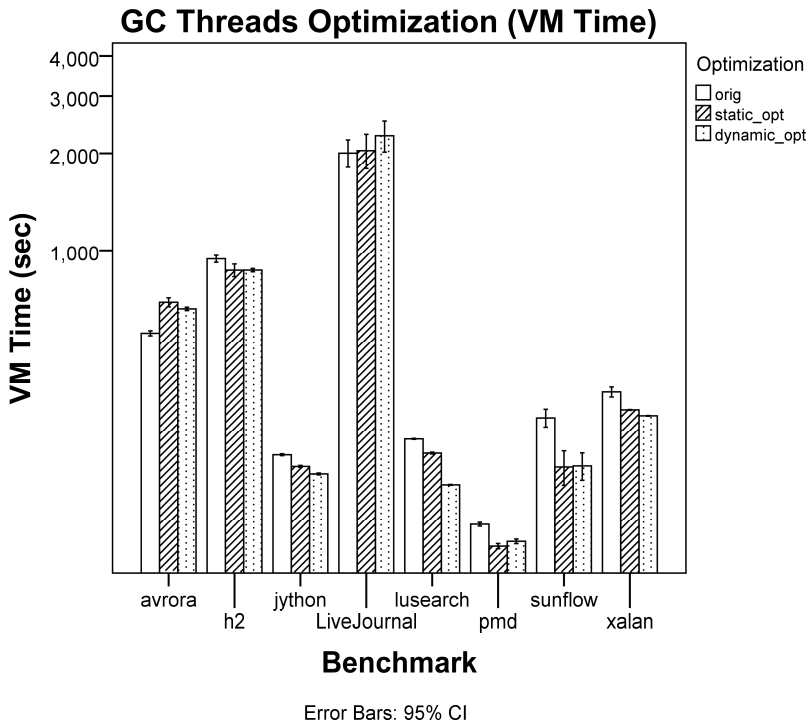


Figure 7.8: Comparison of overall application performance for various garbage collection threading policies (lower is better)

a single JVM instance. The default adaptive garbage collection thread management policy was designed to be responsive to runtime changes, and that inspires us to design a runtime optimization to vary the number of collector threads in the context of NUMA systems. Instead of correlating the number of collector threads with the combination of heap size and the number of Java threads as of present, we take advantage from discussion in previous sections and correlate the number of threads with collection throughput.

Figure 7.9 depicts our runtime optimization system for adaptive garbage collection thread management. When a program pauses execution to enter a garbage collection phase, we activate the adaptive policy to calculate the appropriate number of threads for the current collection cycle. We treat the first and the second garbage collection cycle as a special case because at least two throughput reading are required to identify the direction of collection throughput. Observational results presented in Section 7.3 indicate that the garbage collector uses a small number of threads to collect the heap. Therefore, we set two threads for these collection cycles. Once the garbage collection completes the execution, we calculate and store collection throughput in a circular buffer. For subsequent collections, we calculate median throughput from the buffer and send the result as a parameter to the adaptive policy.

The adaptive policy is a simple search-based runtime optimization called a *gradient descent* [Snyman, 2005] —although with the throughput curves we use, we are actually performing

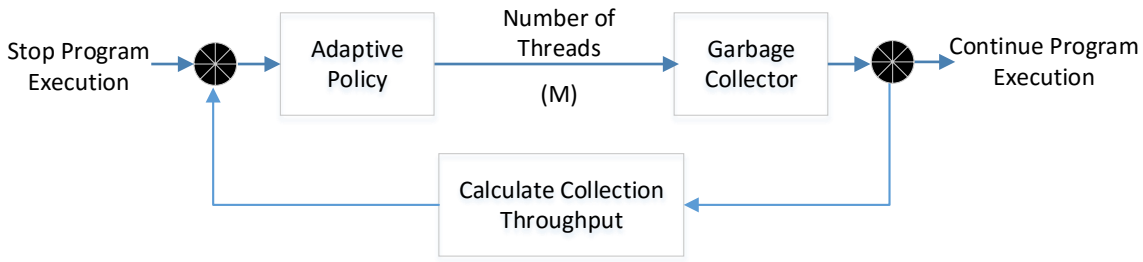


Figure 7.9: Schematic diagram for adaptive runtime GC threads management system.

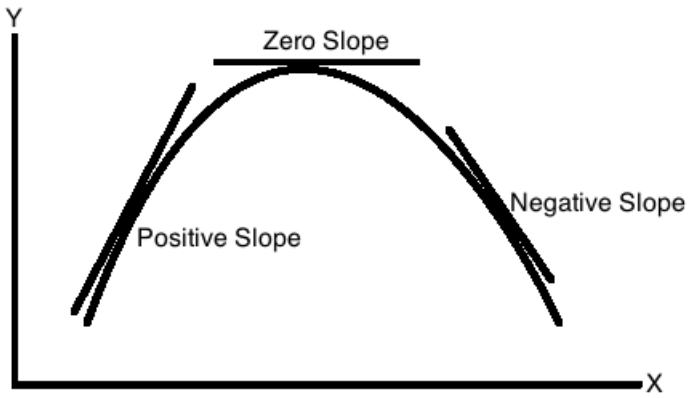


Figure 7.10: Gradient ascent optimization searches for optimal value which is on the top of the hill, where the slope is zero. At any point to the left, the slope value is positive, indicating that optimal direction is forward. If the point is to the right, the slope is negative and the direction is back.

```
1  Δthreads = numThreads − numThreadsAtLastQuery
2  Δthroughput = throughput − throughputAtLastQuery
3
4  numThreadsAtLastQuery = numThreads
5  throughputAtLastQuery = throughput
6
7  if (|Δthroughput| < ϵ)
8  then
9    numThreads does not change  /*below noise threshold*/
10 else
11   numThreads += α * Δthroughput/Δthreads  /*multiply by gradient*/
```

Figure 7.11: Gradient ascent algorithm to optimize the number of GC threads

gradient *ascent*. Figure 7.10 depicts an example for gradient ascent curve. The basic intuition is that, at a point x, we change the number of threads in a way that is proportional to the gradient of the throughput curve at x. In this way, we approach the local maximum for the throughput curve. Figure 7.11 outlines the gradient ascent algorithm we use and Appendix A presents the source code. The parameter ϵ specifies the sensitivity threshold below which any differences are considered to be noise. The parameter α is the amount by which the gradient is multiplied. This value must be set carefully—if α is too small then the optimization takes a long time to converge, but if α is too large then the optimization overshoots the maximum point. In all our experiments, we use the values $\alpha = 0.05$ and $\epsilon = 25$, which are calculated by trial and error effort. As before, we apply the optimization concurrently but separately for minor and major GC threads.

The results in Figure 7.7, for pause time, and Figure 7.8, for total execution time, show that the dynamic selection of collector threads performs *significantly better* than the Hotspot default policy. The gradient ascent approach is generally *as good* as the static optimization technique. The results show 21% on average performance improvement for DaCapo benchmarks. However, note that for some programs (e.g. lusearch) the dynamic approach is *significantly better*. This is likely to be the case when the application goes through different phases within which there are large differences between the optimal number of collector threads. Adversely, LiveJournal does not benefit from the gradient-ascent approach. In fact, the performance degrades by 22% on average. The dynamic approach requires some tuning (for α and ϵ), which we do this once for the system. LiveJournal may need different parameters value than DaCapo benchmarks to gain performance from the dynamic approach.

Figure 7.12 shows how the number of collector threads (for major and minor collections) changes over time for several benchmarks. It shows up to 100 garbage collection cycles for illustration purpose only, though benchmarks use small heap size and call the garbage collection excessively. These are illustrative graphs, but they show the effect of the gradient ascent optimization. Note that the result returned from the gradient ascent approach is bounded in

the range $[2, 32]$.

To sum up, a garbage collector thread management policy based on gradient ascent is able to improve garbage collection performance significantly over the Hotspot default policy, in many cases.

7.6 Related Work

7.6.1 NUMA GC Characterization

Sartor and Eeckhout [2012] study JVM performance for a two-socket NUMA machine. They experiment with running garbage collection threads on one socket at a lower frequency than application threads on the other socket. In general, they find that there can be fewer collector threads than application threads, although they do not expose NUMA congestion as the underlying reason for this.

7.6.2 Causes of Congestion

Gidra et al. [2013] analyzed the OpenJDK Parallel Scavenge garbage collector for parallelism bottlenecks and pinpointed several contended data structures. They propose several modifications to reduce contention (and corresponding NUMA congestion).

Iyengar et al. [2012] study the scalability of the marking phase of the C4 algorithm [Tene et al., 2011]. They report that the duty cycles of the marking phase get worse as the number of threads increases. A primary source of this problem is the contention of work sharing in marking tasks, where multiple threads attempt to atomically update words in a side bitmap. The JVM allocates the bitmap data structure at the initialization phase and it is likely to reside in a single NUMA node. Therefore, the collector threads would saturate the bandwidth of that node. In addition, the cache line is big enough to accommodate multiple words; thus, false sharing would occur with high probability.

Garbage collection's work stealing may also lead to congestion. Gidra et al. [2011] evaluate disabling work stealing and report that some applications gain performance improvement. Muddukrishna et al. [2013] propose a locality-aware work stealing algorithm. Cores in a multi-hop memory hierarchical systems calculate the distance to other NUMA nodes. Threads that run out of work on a node attempt to steal work from the 'nearest' pending queues. Olivier et al. [2011] develop a hierarchical work stealing algorithm to improve locality. In each node, one third of running threads steal work from other nodes on behalf of the remaining threads in the same chip. Stolen work is pushed into a node-local shared queue, which enables threads to consume work from a local queue.

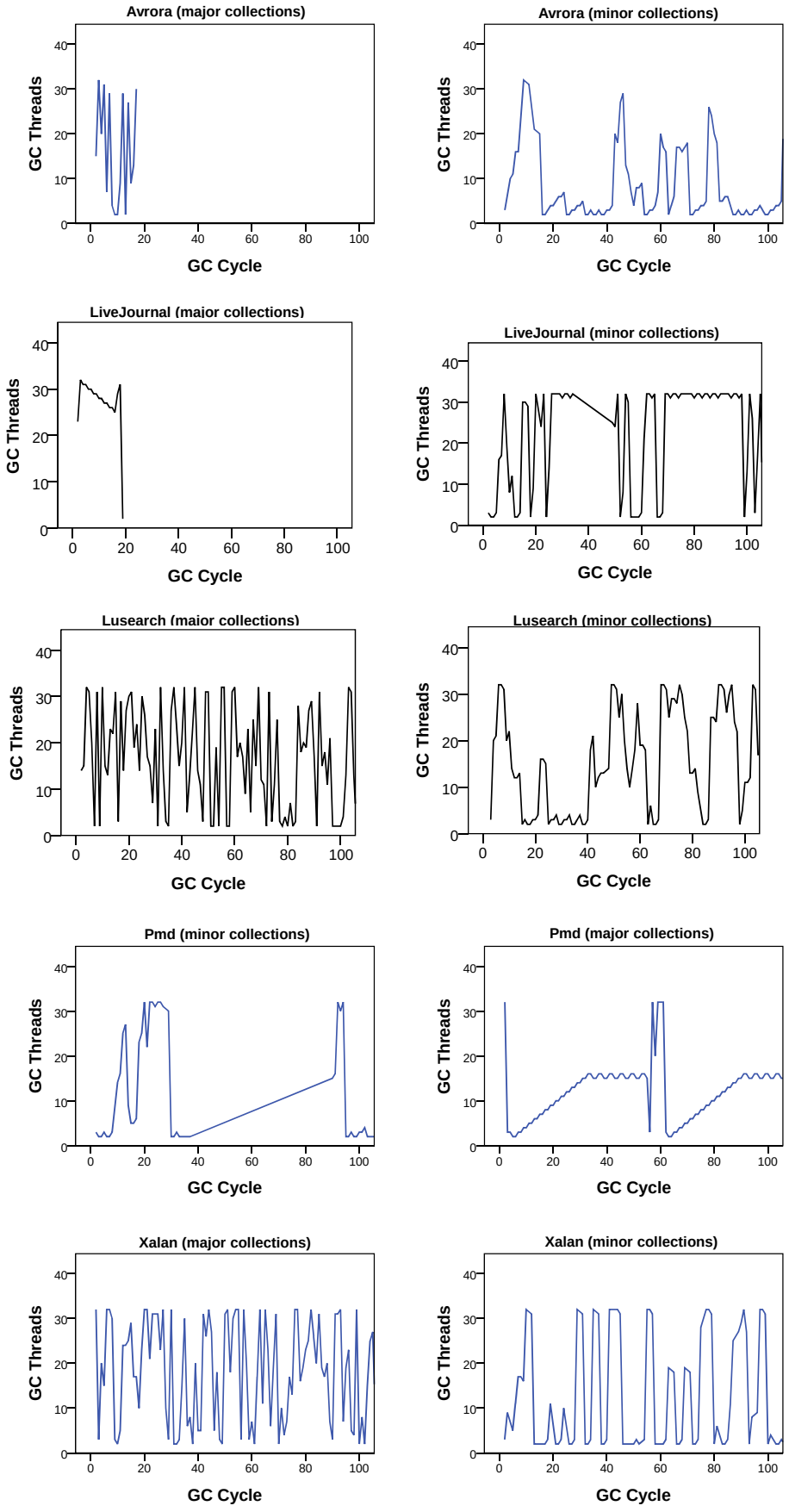


Figure 7.12: Illustrative graphs showing how the number of collector threads varies over the first 100 collections with gradient ascent optimization

7.6.3 Reducing Congestion

Gidra et al. [2015] use message passing between local and remote garbage collection threads to prevent NUMA congestion. Locality improvement is the most common way to reduce congestion. This can be achieved by using thread-local heaps [Marlow and Peyton Jones, 2011, Domani et al., 2002]. Object migration may improve locality, but causes non-trivial inter-node traffic [Tikir and Hollingsworth, 2005].

7.7 Conclusion

In the same way that JVM heap size is subject to a complex dynamic tuning system based on feedback control, we argue that the number of garbage collection threads should have a similar runtime tuning system. We have shown that the number of garbage collection threads has a significant impact on application performance. NUMA machines are particularly sensitive to garbage collection multi-threading. We observe that the optimum number of collector threads varies by garbage collection type and application workload. We have shown the potential for adaptive tuning using a simple search-based optimization technique.

The static optimization identifies the optimal number of threads based on the collection throughput behavior. Previous studies, e.g. [Mao et al., 2009], have shown that the runtime performance of DaCapo Java benchmarks are sensitive to input data variation. Our static optimization is specialized to the particular input data used for training / analysis. For this reason, the dynamic optimization approach is better when input characteristics are not known ahead-of-time.

However, there are several limitations in this study. We only perform experiments on one NUMA system and with a single virtual machine implementation. It is possible that these empirical behaviors might be artifacts of the evaluation platform. To mitigate this, it is necessary to perform the same experiments on other platforms and compare the results. We consider our range of benchmarks to be representative and moderately broad. Furthermore, we could check to see whether Hotspot has similar multi-threading garbage collection performance issues on multicore UMA platforms.

Part III

CONCLUSION

CHAPTER

8

CONCLUSION

Modern servers increasingly use NUMA architectures to increase the number of processing cores and the memory bandwidth. Whilst the parallel hardware in NUMA systems have the potential to improve application performance significantly, balancing NUMA locality and off-node traffic is a challenge. Managed runtime systems abstract NUMA details to provide several software engineering advantages, including automatic memory management. Re-engineering runtime systems to be topology-aware can assist non-specialist program developers to benefit from advanced hardware.

The aim of this dissertation is to investigate improvements to the garbage collection of the JVM by re-visiting the design choices for garbage collection from ground up. The scope of this study includes ccNUMA architectures which are the base architecture for modern servers. Cache coherency protocols have limited the scalability of the number of cores, however, this research has shown that NUMA aspects could degrade the memory access performance if software treats the memory in a NUMA-agnostic fashion. In addition, this study is limited to Hotspot JVM which is the prevalent deployment of JVMs. Garbage collection optimizations in this research target the default garbage collection policy, i.e. parallel scavenge. Other garbage collection policies would be a research extension in the future.

Although NUMA awareness optimizations discussed in this dissertation have shown improvement to the garbage collection, there should be a comprehensive re-visit to the design choices for NUMA garbage collection policies from ground-up to be NUMA aware. This chapter concludes the dissertation by revisiting the thesis statement presented in Chapter 1

and discussing potential extensions and future work.

8.1 Thesis Statement Revisited

The thesis statement presented in Section 1.2 (page 4) is as follows:

> *Given that NUMA systems partition the memory into multiple nodes, and a multi-threaded application can allocate data in any NUMA node, parallel garbage collection involves off-node communications cost when collecting garbage memory. This research asserts that NUMA topology awareness can improve garbage collection performance. By obtaining data location, garbage collection threads can process NUMA-local data. In addition, NUMA congestion caused by increased number of scheduled garbage collection threads can be alleviated by dynamically adapting the number of threads.*

The assertion is proven by the following findings of this research:
Chapter 5 studies reference locality when garbage collection threads traverse the reference graph. Two major techniques play an important role to characterize reference locality: parallel algorithms and memory allocation policies. Garbage collectors are obliged to process the reference graphs created by mutator threads during program execution. The JVM devolves memory allocation to the operating system, which is set to first-touch policy by default and uses huge page tables for page mapping.

Previous research proposals base their approaches on inspecting every object location prior to processing for copying or compacting garbage collectors. This chapter shows that huge page tables map large memory pages to a single NUMA node. Therefore, the study exploits this mapping mechanism and provides empirical evidence that a large number of connected objects belonging to the transitive closure of a root reside in the same NUMA node as that root. The reference graph can be divided into many rooted sub-graphs that garbage collection threads leverage to improve NUMA locality.

Chapter 6 discusses the implementation of NUMA-aware garbage collection based on the observation of rooted sub-graph locality richness. In this chapter, three optimization schemes are presented. Firstly, the aggressive scheme allows garbage collection threads to process NUMA-local rooted sub-graphs only. To implement this scheme, garbage collection threads classify roots based on NUMA nodes and enqueue them into shared per-node queues. Then, each thread dequeues roots from a NUMA-local queue and scans their transitive closures. During the work stealing phase, garbage collection threads steal work from NUMA-local pending queues only. The results show that the aggressive optimization scheme works best for large-heap applications. The average performance gains for aggressive scheme is 13%.

Secondly, the hybrid scheme allows garbage collection threads to process NUMA-local rooted sub-graphs only but enables work stealing from any pending queues. In this scheme, stealing from remote NUMA nodes may reduce NUMA congestion in the local node or balance the workload over NUMA nodes if the memory allocation is biased to some nodes. Small-heap applications benefit greatly from the hybrid optimization scheme. The hybrid scheme improves the garbage collection performance with an average of 23%.

Thirdly, the relaxed scheme aims at preserving the rooted sub-graph integrity by allowing garbage collection threads to process all rooted-sub-graphs prior to entering the work stealing phase. The results show that the relaxed optimization scheme outperforms the default JVM with 8% on average.

The three optimization schemes of the NUMA-aware garbage collector combine to improve the overall collection performance by 15%.

Chapter 7 addresses high NUMA off-node traffic caused by the large number of scheduled garbage collection threads. This chapter explores the relationship between collection throughput and thread count. The study concludes that at a threshold thread count, the collection throughput peaks and then degrades as the number of threads increases. The study is repeated, however, to observe NUMA traffic behavior as more garbage collection threads are added. We use hardware performance counters to measure memory events (read/write) throughout the program execution. The results show that a high correlation between the number of threads and NUMA traffic is noticeable.

Based on these studies, this chapter provides a static optimization scheme by enabling the user to set the appropriate number of garbage collection threads using command-line flags, separately for minor and major collections. The static optimization performs 25% on average better than the default JVM for the majority of benchmarks. In addition, this chapter uses a runtime thread management policy to change the number of threads at each collection cycle based on previous collection throughput behavior. This policy uses a gradient-ascent algorithm to calculate the step of garbage collection thread count required for the current collection. The results show 21% performance improvement for DaCapo benchmarks. LiveJournal benchmark performs worse than the default JVM because the gradient-ascent algorithm's parameters require further tuning to give a different step size from the one used for DaCapo benchmarks.

8.2 Contributions

This thesis contributes to garbage collection in the context of NUMA architectures in several ways:

- **Rooted sub-graph locality** *Chapter 5*
  Unlike previous research, this work develops the rooted sub-graph hypothesis, which requires garbage collector threads to obtain the location of root references only (Section 5.2). The root set size is smaller than the live object set that previous approaches use. To test the hypothesis, we use seven benchmarks from the established real-world DaCapo benchmark suite. In addition, the experiments involve a widely used SPECjbb benchmark and Neo4J graph database Java benchmark, as well as an artificial benchmark. The results presented in Section 5.6 show that 80% of rooted sub-graph objects (in the old generation) reside in the same NUMA node as the root. The young generation's rooted sub-graphs show less locality connection between objects. Section 6.4.2 analysed the cause of this reduction and concluded that roots that are from the old-to young generation incur poor locality. Therefore, NUMA optimizations can be be applied to high-locality rooted sub-graphs only.

- **Root classification** *Chapter 6*
  By using the rooted sub-graph, task generation and distribution can improve NUMA locality. Roots are classified and distributed to NUMA queues, where garbage collector threads can acquire appropriate roots. Section 6.3 describes how roots are classified and distributed to NUMA queues.

- **NUMA-local work stealing** *Chapter 6*
  Work stealing usually selects an arbitrary pending queue to steal work. In NUMA architecture, this mechanism may incur remote memory access overhead and change object location. NUMA-local work stealing ensures that garbage collection threads that run out of work can steal work from NUMA-local pending queues. The implementation of NUMA-local work stealing is presented in Section 6.3. Work stealing should consider two conflicting factors. For locality enhancement, work stealing is limited to local NUMA nodes to avoid remote memory access. However, for load balancing and to reduce congestion in the local NUMA node, garbage collection threads should steal from remote NUMA nodes. This research suggests that managing these factors in adaptive runtime systems is application-specific. Our Aggressive and Hybrid optimization schemes use NUMA-local work stealing and the results show 13% and 23% on average performance improvement, respectively.

- **Collection throughput and NUMA traffic correlate with the number of garbage collection threads** *Chapter 7*
  Selecting the appropriate number of garbage collection threads is a non-trivial problem. In this research, the number of garbage collection threads is correlated with collection throughput. Section 7.3 observes that garbage collection throughput increases

as more threads added to the collector thread pool and peaks at a specific number of garbage collection threads then the collection throughput declines. Likewise, NUMA off-node is shown to be impacted by the number of garbage collection threads. Both factors suggest that the naive utilization of full resources for garbage collection yields suboptimal performance. In addition, the optimal number of threads is shown to be application-specific. Analysis of collection throughput behavior indicates that there is an optimum number of threads for each application and this number is different for minor and major collections (Figure 7.2 and Figure 7.3).

- **NUMA-aware garbage collection thread management** *Chapter 7*
  This research contributes to garbage collection by using static and dynamic thread management. In the static optimization policy, the optimal number of garbage collection threads is obtained for minor and major collection individually for each benchmark (Section 7.4). The static optimization performs 25% better than the default JVM. The dynamic optimization policy uses a search-based algorithm *gradient-ascent* to find optimal points in a quadratic curve, which represents the collection throughput, Section 7.5). At runtime, the number of garbage collection threads changes based on previous collection throughput behavior and the slope direction that the gradient-ascent algorithm calculates. The dynamic optimizations show 21% better performance than the default JVM configuration (Figure 7.7) for DaCapo benchmarks. The gradient-ascent algorithm requires further tuning for LiveJournal benchmark to gain improvement.

The outcomes of this research have shown that garbage collection exhibits performance degradation when threads access remote NUMA nodes in ccNUMA systems. Locality improvements for NUMA memory accesses are shown to be more significant than cache locality improvement in ccNUMA architectures. Therefore, this research contributions would be likely to improve next generations of non cache-coherent NUMA architectures (will be discussed in Section 8.3.2). In addition, rooted sub-graph's locality richness is tested for stop-the-world copying and compacting garbage collections. Although we use Hotspot JVM in this research, we would expect similar results for other JVMs since there is a phase for roots enumeration as described in Section 3.6. The next section identifies areas where these contribution can be applied.

8.3 Future Research Directions

This dissertation has shown that NUMA awareness is significant to garbage collection performance. Developing NUMA-aware algorithms for garbage collection that balance between

data locality and congestion is non-trivial. The work of this thesis shows that there is great potential for future research in this area. This section outlines and describes a number of possible research directions.

8.3.1 Experimental Setup Generalization

So far, this thesis investigates the effectiveness of using NUMA-aware garbage collection under one VM variant (the Hotspot JVM), one operating system (Linux), and one NUMA architecture (AMD Opteron multi-hop memory system). There are many other hardware and software combinations on which NUMA garbage collection can be used to examine and generalize suggested optimizations.

Different Hardware

Hardware advances are leading information technology solutions and services; however, the software needs to increase the development pace to utilize the emerging powerful hardware. All experiments in this dissertation are executed on the AMD Opteron platform, see Chapter 4. The number of cores in a single die, interconnection link transfer speed, cache hierarchy, and cache coherency protocol are all changing rapidly and they have significant impact on NUMA performance.

AMD Opteron processors use the Bulldozer architecture and its successors, for example Piledriver. This architecture has a per-core integer module and shares cache with another sibling core. However, AMD has recently announced the Zen micro-architecture [Wikipedia, 2016], which introduces simultaneous multi-threading, a similar feature offered by Intel processors (Hyper Threading), to improve per-core performance. This micro-architecture may challenge NUMA garbage collection performance because adding more ”virtual” cores would cause congestion on bus and memory controller [Gaud et al., 2015]. The Zen micro-architecture uses high memory bandwidth (HMB) [AMD, 2016a] chips, which is a high-performance RAM interface for 3D-stacked DRAM. In contrast to our experimental hardware which integrates the memory controller with the processor, HMB chips involves an interface with the CPU and may increase the communication overhead.

Different Software

High-level languages use managed runtime systems because of their success in abstracting low-level details. The JVM documentation gives high-level specifications for garbage collection design; thus there are diversity of JVMs production that may implement different garbage collection algorithms, for instance JRockit, J9, Hotspot, Jikes RVM, and Zing

JVMs. Moreover, the JVMs may provide multiple garbage collections policies for different application needs.

Hotspot provides three garbage collection policies: the Parallel Scavenge, the Concurrent Mark-Sweep, and the Garbage First. In this research, NUMA optimization is applied to the Parallel Scavenge garbage collection policy. It would be a fruitful research extension to apply similar NUMA optimizations to the other two garbage collection policies.

Various programming languages can be hosted on JVMs, for example, Scala, Clojure, Jython, and JRuby. Object size and lifetimes of these languages are different from Java [Li et al., 2013]. However, Hotspot condenses long-living objects into one side of the old generation and usually does not move them [Hotspot_dense_prefix] (`compute_dense_prefix` method). Furthermore, functional languages, e.g. Haskell, have a high allocation rate, which may impact NUMA memory allocation and off-node traffic for multi-threaded applications. Our optimizations (Section 6.3) would be candidates for such languages to improve NUMA locality. Extending this research to study NUMA garbage collection for these languages is another research opportunity.

Different Operating System

In this research, the operating system used to run the experiments is Linux (kernel version 3.11). NUMA-aware thread scheduling and memory allocation policies are different from one operating system to another. For example, Solaris uses two modes of memory allocation: next-touch and random [Antony et al., 2006]. Next-touch policy is for thread-local data, whereas random is useful for shared data that multiple threads can access. Data placement is determined by the memory allocation **and** thread affinity. Linux supports more flexibility in memory allocation policies than Solaris. Experimenting with the proposed NUMA optimizations on Solaris or other operating systems may show interesting results.

Future runtime systems may need to override the operating system's memory allocation policy to manage data locality. In this case, garbage collection threads can manage data placement on NUMA architectures, e.g. [Gidra et al., 2013].

8.3.2 NUMA Architectures without Cache Coherency

Cache-coherent shared memory systems enable multicore processors to benefit from private caches and reduced memory access latency and traffic [Choi et al., 2011]. However, the conventional wisdom is that cache coherence protocols will be a major obstacle to scaling the number of cores that future processors are expected to have [Martin et al., 2012, Komuravelli et al., 2014]. Increasing interconnection network traffic would cause higher access latency and power consumption.

Three directions can be taken to solve this problem: abandon hardware and rely on software cache coherence protocols, remain dependent on hardware cache coherency, or use software-hardware hybrid coherency management, which aims to apply hardware cache coherency within a processor chip and software-managed coherency between processors. For example, Barrelfish [Baumann et al., 2009] allows each core to run a separate operating system kernel and communicate with other cores through message passing protocols. Intel's Single-Chip Cloud Computer (SCC) [Baron, 2010] abandons hardware cache coherence and replaces it with message-passing software cache management.

The work in this thesis investigates NUMA optimization on systems with hardware-enabled cache coherency protocols. When moving the coherency burden to software (message passing), my proposed NUMA optimizations would need revisiting, and this too presents an exciting research opportunity.

APPENDIX

A

GRADIENT-ASCENT ALGORITHM

This appendix presents the gradient-ascent class file layout. It shows the header and the method source code.

Listing A.1: Gradient Ascent Header File

```
1  class HillClimber {
2  private:
3    // number of threads at last GC, so we can calculate
       whether we should
4    // recommend increase or decrease in number of GC threads
5    int numThreadsAtLastQuery;
6    // GC throughput reading at last GC, so we can calculate
       whether we are
7    // ascending or descending the hill
8    double throughputAtLastQuery;
9    // noise threshold - if two successive throughput
10   // readings are within epsilon of each other then recommend
        no change
11   // to num GC threads
12   double _epsilon;
```

```
13   // gradient multiplication factor to scale the
14   // gradient ascent - this must be tuned carefully for the
       problem
15   // (a) if alpha is too small then the search takes too long
        to get to optimum
16   // (b) if alpha is too large then the search overshoots
       badly
17   double _alpha;
18 // cached recommendation for num GC threads
19   //    +ve means increase num threads
20   //      0   means keep num threads constant
21   //     -ve means decrease num threads
22   // Notes:
23   //  (1) _directionToClimb is the cached return value of
24   //       the most recent call to recordThroughputReading()
25   //  (2) the magnitude of this value is computed by the
26   //      "gradient ascent" approach - so it can be used to
27   //      determine how many threads to grow/shrink the
28   //      GC thread count.
29   //      _directionToClimb = gradient * alpha
30   int _directionToClimb;
31 public:
32   // constructor
33   // noise threshold and gradient scaling factor
34   // HillClimber(double epsilon, double alpha);
35   HillClimber(double epsilon = 25.0, double alpha = 0.05)
36     : _epsilon(epsilon), _alpha(alpha) {
37     numThreadsAtLastQuery = 2;
38     throughputAtLastQuery = 0.0;
39     _directionToClimb = 0;
40   }
41   // supply a (numThreads, throughput) reading to the hill-
       climber
42   // This method compares the new reading with the previous
       one, and
43   // outputs a recommendation. See comment on
       _directionToClimb
44   // to interpret the recommendation value.
```

```
45   int recordThroughputReading(int numThreads, double
       throughput);
46   // returns cached direction computed at last call to
       recordThroughputReading()
47   // Again, see comment on _directionToClimb to interpret the
        recommendation
48   // value.
49   int directionToClimb();
50   int getLastThreads() {
51     return numThreadsAtLastQuery;
52   }
53 };
```

Listing A.2: Gradient Ascent Methods

```
1 // returns the new direction to go...
2 // (-ve means decrease numThreads,
3 //  +ve means increase numThreads,
4 //  0 means keep numThreads constant)
5 // magnitude of returned value computed by
6 // gradient ascent - larger for steeper gradients
7 int
8 HillClimber::recordThroughputReading(int numThreads, double
    throughput) {
9   int delta_threads = numThreads - numThreadsAtLastQuery;
10  double delta_throughput = throughput -
      throughputAtLastQuery;
11  bool ignoreChange = fabs(delta_throughput) < _epsilon;
12  double gradient = (delta_threads==0)?0:(delta_throughput /
      delta_threads);
13  //std::cout << "[hillclimber] gradient is " << gradient <<
      "\n";
14  // now update fields
15  numThreadsAtLastQuery = numThreads;
16  throughputAtLastQuery = throughput;
17  if (ignoreChange) {
18    // stay in same place on hill
```

```
19      _directionToClimb = 0;
20    } else {
21      _directionToClimb = ((gradient<0)?floor(gradient):ceil(
           gradient)) * _alpha;
22    }
23      if (delta_threads == 0) {
24       if (numThreads < 16) {
25            _directionToClimb = 1;
26          } else {
27             _directionToClimb =-1;
28          }
29      }
30    //std::cout << "[hillclimber] recommending change: " <<
         _directionToClimb << "\n";
31    return _directionToClimb;
32  }
33  // returns the cached most recently computed
34  //  direction to go...
35  // (-ve means decrease numThreads,
36  //  +ve means increase numThreads,
37  //  0 means keep numThreads constant)
38  int
39  HillClimber::directionToClimb() {
40    return _directionToClimb;
41  }
```

BIBLIOGRAPHY

Khaled Alnowaiser. A Study of Connected Object Locality in NUMA Heaps. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 1:1–1:9, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2917-0. doi: 10.1145/2618128.2618132. URL http://doi.acm.org/10.1145/2618128.2618132.

Khaled Alnowaiser and Jeremy Singer. *Topology-Aware Parallelism for NUMA Copying Collectors*, chapter Languages and Compilers for Parallel Computing: 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers, pages 191–205. Springer International Publishing, Cham, 2016. ISBN 978-3-319-29778-1. doi: 10.1007/978-3-319-29778-1_12. URL http://dx.doi.org/10.1007/978-3-319-29778-1_12.

AMD. AMD64 Architecture Programmers Manual Volume 2: System Programming. http://support.amd.com/TechDocs/24593.pdf, Sep 2015.

AMD. High Bandwidth Memory — Reinventing Memory Technology. http://www.amd.com/en-us/innovations/software-technologies/hbm, June 2016a.

AMD. HyperTransport Technology. http://www.amd.com/en-us/innovations/software-technologies/hypertransport, June 2016b.

Todd A. Anderson. Optimizations in a Private Nursery-based Garbage Collector. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 21–30, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0054-4. doi: 10.1145/1806651.1806655. URL http://doi.acm.org/10.1145/1806651.1806655.

David L Andre. Paging in lisp programs. Master's thesis, University of Maryland, 1986.

Kleen Andreas. A NUMA API for LINUX. http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf, April 2005.

Joseph Antony, Pete P. Janes, and Alistair P. Rendell. *High Performance Computing - HiPC 2006: 13th International Conference, Bangalore, India, December 18-21, 2006. Proceedings*, chapter Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport, pages 338–352. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-68040-6. doi: 10.1007/11945918_35. URL http://dx.doi.org/10.1007/11945918_35.

N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001. ISSN 14324350. doi: 10.1007/s002240011004.

Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage Collection for Multicore NUMA Machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 51–57, New York, NY, USA, 2011. ACM. ISBN 9781450307949. doi: 10.1145/1988915.1988929. URL http://portal.acm.org/citation.cfm?doid=1988915.1988929.

David F. Bacon and V. T. Rajan. *Concurrent Cycle Collection in Reference Counted Systems*, chapter Concurrent Cycle Collection in Reference Counted Systems, pages 207–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-45337-6. doi: 10.1007/3-540-45337-7{_}12. URL http://link.springer.com/10.1007/3-540-45337-7{_}12.

DavidF. Bacon, StephenJ. Fink, and David Grove. Space- and Time-Efficient Implementation of the Java Object Model. In Boris Magnusson, editor, *ECOOP 2002 Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 111–132. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43759-8. doi: 10.1007/3-540-47993-7_5. URL http://dx.doi.org/10.1007/3-540-47993-7_5.

Katherine Barabash and Erez Petrank. Tracing Garbage Collection on Highly Parallel Platforms. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 1–10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0054-4. doi: 10.1145/1806651.1806653. URL http://doi.acm.org/10.1145/1806651.1806653.

Max Baron. The Single-chip Cloud Computer . http://http://www.intel.co.uk/content/dam/www/public/us/en/documents/

technology-briefs/intel-labs-single-chip-cloud-article.pdf, Apr 2010.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629579. URL http://doi.acm.org/10.1145/1629575.1629579.

E D Berger, K S McKinley, R D Blumofe, and P R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *Acm Sigplan Notices*, 35(11):117–128, 2000. ISSN 0362-1340. doi: 10.1145/356989.357000.

Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 344–358, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. doi: 10.1145/949305.949336. URL http://doi.acm.org/10.1145/949305.949336.

Stephen M Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S McKinely, and J Eliot B Moss. Pretenuring for java. In *ACM SIGPLAN Notices*, volume 36, pages 342–352. ACM, 2001.

Stephen M. Blackburn, Richard E. Jones, Kathryn S. Mckinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. *PLDI: Programming Language Design and Implementation*, volume37(5):153–164, 2002. doi: 10.1145/512529.512548.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The Dacapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. URL http://doi.acm.org/10.1145/1167473.1167488.

Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish

Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, August 2008. ISSN 0001-0782. doi: 10.1145/1378704.1378723. URL http://doi.acm.org/10.1145/1378704.1378723.

Hans Boehm. GCBench Program. http://hboehm.info/gc/gc_bench/GCBench.java, Oct 2000.

William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. Numa policies and their relation to memory architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 212–221, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: 10.1145/106972.106994. URL http://doi.acm.org/10.1145/106972.106994.

François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010. doi: 10.1109/PDP.2010.67. URL https://hal.inria.fr/inria-00429889.

Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-Conscious Data Placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 139–149, New York, NY, USA, 1998. ACM. ISBN 1-58113-107-0. doi: 10.1145/291069.291036. URL http://doi.acm.org/10.1145/291069.291036.

Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 12–24, New York, NY, USA, 1994. ACM. ISBN 0-89791-660-3. doi: 10.1145/195473.195485. URL http://doi.acm.org/10.1145/195473.195485.

Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 332–340, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134021. URL http://doi.acm.org/10.1145/1133981.1134021.

C J Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13 (11):677–678, 1970a.

C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Commun. ACM*, 13(11):677–678, November 1970b. ISSN 0001-0782. doi: 10.1145/362790.362798. URL http://doi.acm.org/10.1145/362790.362798.

Yannis Chicha and Stephen M Watt. A Localized Tracing Scheme Applied to Garbage Collection. *Programming Languages and Systems*, 4279:323–339, 2006. doi: 10.1007/11924661_20.

TM Chilimbi and JR Larus. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. *ACM SIGPLAN Notices*, 1(212), 1999. URL http://dl.acm.org/citation.cfm?id=286865.

Trishul M. Chilimbi and James R. Larus. Using Generational Garbage Collection to Implement Cache-conscious Data Placement. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 37–48, New York, NY, USA, 1998. ACM. ISBN 1-58113-114-3. doi: 10.1145/286860.286865. URL http://doi.acm.org/10.1145/286860.286865.

Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious Structure Definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 13–24, New York, NY, USA, 1999a. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301635. URL http://doi.acm.org/10.1145/301618.301635.

Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999b. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301633. URL http://doi.acm.org/10.1145/301618.301633.

B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 155–166, Oct 2011. doi: 10.1109/PACT.2011.21.

Myra Cohen, Shiu Beng Kooi, and Witawas Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1901–1908, New York, NY, USA, 2006. ACM. ISBN 1-59593-186-4. doi: 10.1145/1143997.1144314. URL http://doi.acm.org/10.1145/1143997.1144314.

George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12): 655–657, December 1960. ISSN 0001-0782. doi: 10.1145/367487.367501. URL http://doi.acm.org/10.1145/367487.367501.

P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *Micro, IEEE*, 27 (2):10–21, March 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.43.

Robert Courts. Improving Locality of Reference in a Garbage-collecting Memory Management System. *Commun. ACM*, 31(9):1128–1138, September 1988. ISSN 0001-0782. doi: 10.1145/48529.48536. URL http://doi.acm.org/10.1145/48529.48536.

E.H.M. Cruz, M.A.Z. Alves, and P.O.A. Navaux. Process mapping based on memory access traces. In *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*, pages 72–79, Oct 2010. doi: 10.1109/WSCAD-SCC.2010.26.

M Dashti, Alexandra Fedorova, and Justin Funston. Traffic management: a holistic approach to memory placement on NUMA systems. *...and operating systems*, pages 381–393, 2013. URL http://dl.acm.org/citation.cfm?id=2451157.

Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 345–357, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349344. URL http://doi.acm.org/10.1145/349299.349344.

Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158611. URL http://doi.acm.org/10.1145/158511.158611.

Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local Heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 76–87, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. doi: 10.1145/512429.512439. URL http://doi.acm.org/10.1145/512429.512439.

Toshio Endo, K. Taura, and A Yonezawa. A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–14, San Jose, CA, 1997. ISBN 0897919858.

Haggai Eran and E Petrank. A Study of Data Structures with a Deep Heap Shape. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 21–28, Seattle, Washington, 2013. ACM. ISBN 9781450312196. URL http://dl.acm.org/citation.cfm?id=2492413.

Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012. ISSN 02721732. doi: 10.1109/MM.2012.17.

CH Flood, David Detlefs, N Shavit, and X Zhang. Parallel Garbage Collection for Shared Memory Multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, page 21, Monterey, California, 2001. USENIX Association. URL https://www.usenix.org/event/jvm01/full_papers/flood/flood.pdf.

M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA systems. *Commun. ACM*, 58(12):59–66, 2015. doi: 10.1145/2814328.

P. Gepner and M.F. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 0–4, 2006. doi: 10.1109/PARELEC.2006.54.

Lokesh Gidra, Gael Thomas, Julien Sopena, and Marc Shapiro. Assessing the Scalability of Garbage Collectors on Many Cores. In *6th Workshop on Programming Languages and Operating Systems (PLOS'11).*, pages 1–7, 2011.

Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451142. URL http://doi.acm.org/10.1145/2451116.2451142.

Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 661–673, 2015.

James Gosling and Alex Buckley. The Java Language Specification: Java SE 8 Edition. *Oracle*, 2015.

Samuel Z. Guyer and Kathryn S. McKinley. Finding Your Cronies: Static Analysis for Dynamic Object Colocation. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 237–250, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: 10.1145/1028976.1028996. URL http://doi.acm.org/10.1145/1028976.1028996.

Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1029–1036, New York, NY, USA, 2005. ACM. ISBN 1-59593-010-8. doi: 10.1145/1068009.1068184. URL http://doi.acm.org/10.1145/1068009.1068184.

Barry Hayes. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 33–46, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi: 10.1145/117954.117957. URL http://doi.acm.org/10.1145/117954.117957.

John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*, chapter Memory Hierarchy Design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011a. ISBN 012383872X, 9780123838728.

John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011b. ISBN 012383872X, 9780123838728.

Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the Connectivity of Heap Objects. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 36–49, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. doi: 10.1145/512429.512435. URL http://doi.acm.org/10.1145/512429.512435.

Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based Garbage Collection. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 359–373, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. doi: 10.1145/949305.949337. URL http://doi.acm.org/10.1145/949305.949337.

Hotspot_dense_prefix. Method: compute_dense_prefix. src/share/vm/gc_implementation/shared/adaptiveSizePolicy.cpp.

Hotspot_Source_Code. Method: calc_default_active_workers. src/share/vm/gc_implementation/parallelScavenge/psParallelCompact.cpp.

Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: 10.1145/1028976.1028983. URL http://doi.acm.org/10.1145/1028976.1028983.

RichardL. Hudson and J.EliotB. Moss. Incremental collection of mature objects. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 388–403. Springer Berlin Heidelberg, 1992. ISBN 978-3-540-55940-5. doi: 10.1007/BFb0017203. URL http://dx.doi.org/10.1007/BFb0017203.

Intel. Intel QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, June 2016.

Balaji Iyengar, Edward Gehringer, Michael Wolf, and Karthikeyan Manivannan. Scalable Concurrent and Parallel Mark. In *Proceedings of the 2012 international symposium on Memory Management - ISMM 12*, pages 61–72. ACM, 2012.

R. Jones and A.C. King. A Fast Analysis for Thread-local Garbage Collection with Dynamic Class Loading. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, 2005. ISBN 0769522920. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1541165.

Richard Jones and Chris Ryder. Garbage Collection Should Be Lifetime Aware. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems ({ICOOOLPS}'2006)*, page 8, 2006. URL http://www.cs.kent.ac.uk/pubs/2006/2376/.

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. doi: 1420082795,9781420082791.

Richard E. Jones and Chris Ryder. A Study of Java Object Demographics. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375652. URL http://doi.acm.org/10.1145/1375634.1375652.

Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 335–354, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384641. URL http://doi.acm.org/10.1145/2384616.2384641.

Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the complexity of hardware cache coherence and some implications. *ACM Trans. Archit. Code Optim.*, 11 (4):37:1–37:22, December 2014. ISSN 1544-3566. doi: 10.1145/2663345. URL http://doi.acm.org/10.1145/2663345.

Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. URL http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf.

Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object Type Directed Garbage Collection To Improve Locality. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 404–425, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-55940-X. URL http://dl.acm.org/citation.cfm?id=645648.664829.

Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 11 (7):40:40–40:51, July 2013. ISSN 1542-7730. doi: 10.1145/2508834.2513149. URL http://doi.acm.org/10.1145/2508834.2513149.

Richard P. LaRowe, Carla Schlatter Ellis, and Laurence S. Kaplan. The robustness of NUMA memory management. *ACM SIGOPS Operating Systems Review*, 25(5):137–151, 1991. ISSN 01635980. doi: 10.1145/121133.121158.

Peeter Laud. Analysis for object inlining in java. In *In Proceedings of the Joses Workshop*, pages 1–8, 2001.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, jun 2014.

Ondej Lhotàk and Laurie Hendren. Run-time evaluation of opportunities for object inlining in java. *Concurrency and Computation: Practice and Experience*, 17(5-6):515–537, 2005. ISSN 1532-0634. doi: 10.1002/cpe.848. URL http://dx.doi.org/10.1002/cpe.848.

Wing Hang Li, David R White, and Jeremy Singer. Jvm-hosted languages: they talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 101–112. ACM, 2013.

LinuxMemPolicy. What is Linux Memory Policy? `https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt`, May 2015.

Mengxiao Liu, Weixing Ji, Zuo Wang, and Xing Pu. A memory access scheduling method for multi-core processor. In *Computer Science and Engineering, 2009. WCSE '09. Second International Workshop on*, volume 1, pages 367–371, Oct 2009. doi: 10.1109/WCSE.2009.689.

Zoltan Majo and Thomas R. Gross. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proceedings of the international symposium on Memory management - ISMM '11*, page 11, New York, NY, USA, 2011. ACM. ISBN 9781450302630. doi: 10.1145/1993478.1993481. URL `http://portal.acm.org/citation.cfm?doid=1993478.1993481`.

S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 50–59, 1999. doi: 10.1109/ICSM.1999.792498.

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

Simon Marlow and Simon Peyton Jones. Multicore Garbage Collection with Local Heaps. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 21–32, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0263-0. doi: 10.1145/1993478.1993482. URL `http://doi.acm.org/10.1145/1993478.1993482`.

Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012. ISSN 0001-0782. doi: 10.1145/2209249.2209269. URL `http://doi.acm.org/10.1145/2209249.2209269`.

J. Harold McBeth. Letters to the editor: On the reference counter method. *Commun. ACM*, 6 (9):575–, September 1963. ISSN 0001-0782. doi: 10.1145/367593.367649. URL `http://doi.acm.org/10.1145/367593.367649`.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM. ISBN 1-58113-741-9. doi: 10.1145/977091.977115. URL http://doi.acm.org/10.1145/977091.977115.

David A. Moon. Garbage Collection in a Large LISP System. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802040. URL http://doi.acm.org/10.1145/800055.802040.

Gordon Moore. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56, 2000.

Ananya Muddukrishna, PeterA. Jonsson, Vladimir Vlassov, and Mats Brorsson. Locality-aware Task Scheduling and Data Distribution on NUMA Systems. In AlistairP. Rendell, BarbaraM. Chapman, and MatthiasS. Mller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40697-3.

Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

Neo4J, March 2015. URL http://www.neo4j.com/.

Gene Novark, Trevor Strohman, and Emery Berger. Custom Object Layout for Garbage-Collected Languages. *Techreport*, (UMCS TR-2006-007), 2006.

Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. A New Approach to Parallelising Tracing Algorithms. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 10–19, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-347-1. doi: 10.1145/1542431.1542434. URL http://doi.acm.org/10.1145/1542431.1542434.

Takeshi Ogasawara. Numa-aware Memory Manager with Dominant-thread-based Copying GC. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 377–390, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640117. URL http://doi.acm.org/10.1145/1640089.1640117.

Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, and Jan F Prins. Scheduling Task Parallelism on Multi-Socket Multicore Systems. *Runtime and Operating Systems for Supercomputers*, pages 49–56, 2011. doi: 10.1145/1988796.1988804.

OpenJDK. OpenJDK repository. `http://hg.openjdk.java.net/jdk8u`, Sep 2015.

Oracle. Java Platform, Standard Edition Tools Reference. `http://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html`, Feb 2016.

Oracle. Hotspot Code. `http://hg.openjdk.java.net/jdk8u/jdk8u20/hotspot/file/f06c7b654d63/src/share/vm/gc_implementation/shared/mutableNUMASpace.hpp`, June 2016a.

Oracle. Java HotSpot Virtual Machine Performance Enhancements. `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.htm`, June 2016b.

Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):384–409, October 1997. ISSN 1084-4309. doi: 10.1145/268424.268464. URL `http://doi.acm.org/10.1145/268424.268464`.

Tony Printezis. Number of Parallel GC Threads. `http://mail.openjdk.java.net/pipermail/hotspot-gc-dev/2009-January/000718.html`, Jan 2009.

T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*, chapter Parallel Computer Architecture. Springer, 2010a. ISBN 9783642048173.

T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*, chapter Message-Passing Programming. Springer, 2010b. ISBN 9783642048173.

Kenneth H. Rosen. *Discrete Mathematics and Its Applications*, chapter Graphs. McGraw-Hill Higher Education, 4th edition, 1999. ISBN 0072424346.

Jennfer Sartor and Lieven Eeckhout. Exploring Multi-threaded Java Application Performance on Multicore Hardware. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*, pages 281—-296, New York, USA, 2012. ACM. ISBN 9781450315616. doi: 10.1145/2384616.2384638. URL `http://dl.acm.org/citation.cfm?doid=2384616.2384638`.

Robert A Saunders. The lisp system for the q-32 computer. *The Programming Language LISP: Its Operation and Applications*, pages 220–231, 1964.

Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting Prolific Types for Memory Management and Optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 295–306, New York, NY, USA, 2002a. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503300. URL http://doi.acm.org/10.1145/503272.503300.

Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 13–25, New York, NY, USA, 2002b. ACM. ISBN 1-58113-471-1. doi: 10.1145/582419.582422. URL http://doi.acm.org/10.1145/582419.582422.

Fridtjof Siebert. Limits of Parallel Marking Garbage Collection. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 21–29, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375638. URL http://doi.acm.org/10.1145/1375634.1375638.

Jeremy Singer. Number of Parallel GC Threads. https://github.com/jeremysinger/gcbench, Feb 2014.

Jeremy Singer, Gavin Brown, Mikel Luján, and Ian Watson. Towards Intelligent Analysis Techniques for Object Pretenuring. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 203–208, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. doi: 10.1145/1294325.1294353. URL http://doi.acm.org/10.1145/1294325.1294353.

Jan A. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*, chapter Line Search Descent Methods for Unconstrained Minimization, pages 33–55. Springer US, Boston, MA, 2005. ISBN 978-0-387-24349-8. doi: 10.1007/0-387-24349-6_2. URL http://dx.doi.org/10.1007/0-387-24349-6_2.

SPEC05. Standard Performance Evaluation Corporation. Available at: http://www.spec.org/jbb2005. URL http://www.spec.org/jbb2005.

SPEC13. Standard Performance Evaluation Corporation. Available at: http://www.spec.org/jbb2013. URL http://www.spec.org/jbb2013.

James W. Stamos. Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. *ACM Trans. Comput. Syst.*, 2(2):155–180, May 1984. ISSN 0734-2071. doi: 10.1145/190.194. URL http://doi.acm.org/10.1145/190.194.

Bjarne Steensgaard. Thread-specific Heaps for Multi-threaded Programs. In *Proceedings of the 2Nd International Symposium on Memory Management*, ISMM '00, pages 18–24, New York, NY, USA, 2000. ACM. ISBN 1-58113-263-8. doi: 10.1145/362422.362432. URL http://doi.acm.org/10.1145/362422.362432.

Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 370–381, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7. doi: 10.1145/320384.320425. URL http://doi.acm.org/10.1145/320384.320425.

SunMicroSystems. Memory Management in the Java HotSpot Virtual Machine. http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf, April 2006.

Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095421. URL http://doi.acm.org/10.1145/1095408.1095421.

Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The Continuously Concurrent Compacting Collector. *Ismm'11*, pages 79–88, 2011. ISSN 0362-1340. doi: 10.1145/1993478.1993491. URL http://www.azulsystems.com/sites/default/files/images/c4_paper_acm.pdf.

Mustafa M. Tikir and Jeffery K. Hollingsworth. Numa-aware Java Heaps for Server Applications. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05, pages 108.2–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9. doi: 10.1109/IPDPS.2005.299. URL http://dx.doi.org/10.1109/IPDPS.2005.299.

Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216, 2010.

D.N. Truong, Francois Bodin, and A. Seznec. Improving Cache Behavior of Dynamically Allocated Data Structures. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 322–329, Oct 1998. doi: 10.1109/PACT.1998.727268.

David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167,

New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. doi: 10.1145/800020.808261. URL http://doi.acm.org/10.1145/800020.808261.

David Ungar and Frank Jackson. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(1): 1–27, 1992.

Andrs Vajda. *Programming Many-Core Chips*, chapter Many-core Virtualization and Operating Systems. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 1441997385, 9781441997388.

Ronald Veldema, Ceriel J. H. Jacobs, Rutger F. H. Hofman, and Henri E. Bal. Object combining: a new aggressive optimization for object intensive programs. *Concurrency and Computation: Practice and Experience*, 17(5-6):439–464, 2005. ISSN 1532-0634. doi: 10.1002/cpe.836. URL http://dx.doi.org/10.1002/cpe.836.

Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 279–289, New York, NY, USA, 1996. ACM. ISBN 0-89791-767-7. doi: 10.1145/237090.237205. URL http://doi.acm.org/10.1145/237090.237205.

Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 217–218, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1563-0. doi: 10.1145/2384716.2384777. URL http://doi.acm.org/10.1145/2384716.2384777.

Wikipedia. Zen (microarchitecture). https://en.wikipedia.org/wiki/Zen_(microarchitecture), March 2016.

Paul R. Wilson. Uniprocessor garbage collection techniques. *Memory Management*, pages 1–42, 1992. doi: 10.1007/BFb0017182.

Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "Static-graph" Reorganization to Improve Locality in Garbage-collected Systems. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 177–191, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113461. URL http://doi.acm.org/10.1145/113445.113461.

Christian Wimmer and Hanspeter Mössenböck. Automatic object colocation based on read barriers. In DavidE. Lightfoot and Clemens Szyperski, editors, *Modular Programming*

Languages, volume 4228 of *Lecture Notes in Computer Science*, pages 326–345. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-40927-4. doi: 10.1007/11860990_20. URL http://dx.doi.org/10.1007/11860990_20.

Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the java hotspot virtual machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 12–21, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254813. URL http://doi.acm.org/10.1145/1254810.1254813.

Ming Wu and Xiao-Feng Li. Task-pushing: A Scalable Parallel GC Marking Algorithm without Synchronization Operations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL http://doi.acm.org/10.1145/216585.216588.

Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why nothing matters: The impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 307–324, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048092. URL http://doi.acm.org/10.1145/2048066.2048092.

Zoe C. H. Yu, Francis C. M. Lau, and Cho-Li Wang. Object co-location and memory reuse for java programs. *ACM Trans. Archit. Code Optim.*, 4(4):4:1–4:36, January 2008. ISSN 1544-3566. doi: 10.1145/1328195.1328199. URL http://doi.acm.org/10.1145/1328195.1328199.

Jin Zhou and Brian Demsky. Memory Management for Many-core Processors with Software Configurable Locality Policies. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1350-6. doi: 10.1145/2258996.2259000. URL http://doi.acm.org/10.1145/2258996.2259000.