공학박사학위논문

Optimizing Memory Management Systems for High Performance and Scalability

높은 성능과 확장성을 위한 메모리 관리 시스템 최적화

2019 년 8 월

서울대학교 대학원

전기·컴퓨터 공학부

박 성 재

Abstract

One common characteristic of modern workloads which appeared with recent computing paradigms including cloud, big data and machine learning is memory intensiveness. Such workloads usually have huge working sets that cannot be fully accommodated in DRAM in many case. Those also tend to show only low locality so that the small CPU cache cannot hide DRAM or lower level memory access overhead.

Meanwhile, computing hardware has also evolved to keep pace with this change. (1) Computing systems are increasing the size of their main memory so that those could accommodate more of the huge working sets. As a result, data center servers utilizing few hundreds of gigabytes of DRAM have been common and even terabytes of DRAM equipped systems exist. (2) Massive parallelism is becomming common and essential. CPU vendors have started to increase the number of CPU cores instead of the CPU frequency due to the heat dissipation and power consumption problem since the early 2000s. Prevalent datacenter systems provide few hundreds of CPU cores; Few thousands of CPU cores are not rare. Such many-core systems are normally constructed in non-uniform memory access (NUMA) architecture. Therefore, efficient, effective and NUMA-awared use of this parallelism is especially important for the memory intensive workloads.

Compared to these rapid changes of workload characteristics and hardware, memory management system software has not sufficiently optimized. Consequently, the memory management system software has been a bottleneck. In other words, the memory intensive modern workloads cannot fully utilize the evolved modern hardware unless the underlying memory management system is completely optimized. This paper provides an overview of a few limitations in existing memory manage-

ment systems and introduces two optimization approaches for high performance and scalability of the memory management systems. The first approach improves the performance of the memory systems by guaranteeing huge page utilization under memory fragmentation situation. For the guarantee, we introduce a contiguous memory allocator that guarantees success and low latency of its allocations. The second approach intends to optimize the NUMA-aware system scalability. For that, we optimize virtual memory address space management system by substituting virtual memory area (VMA) managing red-black tree protection from global reader-writer locking to an RCU extension. Because no RCU extension including state-of-the-arts are NUMA oblivious, we also designed new RCU extension that provides NUMA-aware scalable update-side synchronization.

Keywords: Multicore, Parallelism, RCU, Fragmentation, Memory, Operating System
Student Number: 2012-23213

Contents

Abstract **1**

Chapter 1 Introduction **6**

1.1 Motivation 6

1.2 Approaches 7

1.2.1 An Optimization for High Performance 7

1.2.2 An Optimization for High Scalability 9

1.3 Dissertation Structure 10

Chapter 2 Guaranteed Transparent Huge Pages Allocations **12**

2.1 Introduction 12

2.2 Background 16

2.2.1 Devices using DMA 16

2.2.2 Huge Pages 17

2.2.3 Buddy Allocator 20

2.2.4 Memory Reservation 21

2.2.5 Contiguous Memory Allocator 21

2.3 Guaranteed CMA 22

2.3.1 Secondary Class Clients of GCMA 23

2.3.2 Limitations and Optimizations 26
2.4 Implementation 27
2.4.1 Contiguous Memory Allocation 29
2.4.2 DMEM: Discardable Memory 30
2.5 Guaranteed THP 30
2.6 Evaluation 32
2.6.1 Evaluation on a Mobile System 32
2.6.2 Evaluation on a Server System 38
2.7 Related Work 45
2.8 Conclusion 47

Chapter 3 A Scalable Virtual Address Space Protected by an HTM-based NUMA-aware RCU Extension 48

3.1 Introduction 48
3.2 Background and Related Work 50
3.2.1 Read-Copy Update 50
3.2.2 Hardware Transactional Memory 53
3.2.3 Related Work 54
3.3 An RCU Extension for NUMA Systems 57
3.3.1 Root Cause of HTM Performance Degradation on NUMA systems 57
3.3.2 Design of RCX 62
3.3.3 Implementation 70
3.4 Evaluation 71
3.4.1 Evaluation Setup 71
3.4.2 Micro-benchmarks 72
3.4.3 Macro-benchmark 76
3.5 Conclusion 80

Chapter 4 Conculsion **81**

Chapter 1

Introduction

1.1 Motivation

Advance of modern computing environments such as cloud, big data, and machine learning is widely spreading memory intensive workloads. In detail, such workloads tend to have huge working sets and low locality, which are not usual with other traditional workloads [1]. Fortunately, two common trend of modern systems hardware could be useful for handling of these workloads.

(1) Modern systems usually equips tons of main memory. Nowadays, data center servers utilizing even few tera bytes of DRAM are not rare and a major processor manufacturer is increasing the level of page tables to efficiently deal with it [2]. Moreover, storage devices are continuously evolving their capacity and speed so that those of high-end devices in these days could be even comparable with that of HDD and DRAM [3, 4]. Thus, at least important parts of the huge working sets of the modern workloads will be able to be accommodated in the main memory.

(2) Massive CPU parallelism is widely spread in modern systems. Due to the heat

dissipation and power consumption problem, since about a couple of decades ago, CPU vendors stopped to increase a frequency of single CPU and started to increase the number of CPU cores in single system. This trend forced software programmers to develop software utilizing massive concurrency so that it can take benefit from these parallel CPU cores. The modern computing environments including cloud, big data, and machine learning also has high concurrency by default owing to their distributed computing based design. Because the performance of each single CPU for the modern workloads tend to bounded by DRAM latency due to their low locality, this parallelism is essential for their performance.

Nevertheless, most existing memory management systems are not designed with aforementioned modern situation that led memory intensive workloads common. Unless this underlying system software for the memory management is fully optimized for this modern trend, the increase of the size of main memory and the number of parallel CPU cores cannot fully optimize the performance of these workloads.

This paper therefore finds out such problematic parts of existing memory management systems that disables high performance and scalability of the modern memory intensive workloads. Based on the findings, this paper designs and introduces two optimization approaches that respectively improves performance and scalability of the memory management systems.

1.2 Approaches

1.2.1 An Optimization for High Performance

The first approach introduces a contiguous memory allocator that guarantees fast and successful allocation, which is especially useful for systems utilizing huge amount of main memory. We call this guaranteed contiguous memory allocator (GCMA).

Though the invention of the virtual memory system has eliminated most every

	Memory utilization	Allocation speed	Fragmentation tolerance
Buddy	High	Very fast	Weak
Reserved	Low	Fast	Strong
CMA	High	Slow	Strong
GCMA	High	Fast	Strong

Table 1.1 Comparison of contiguous memory allocators.

requirement for physically contiguous memory, still many areas including devices having no MMU-like peripherals for themselves and various kernel level purposed buffers require that. Existing physically contiguous memory allocators are based on reservation (Reserved), migration (CMA), or buddy algorithm (Buddy). Because of that, every existing allocator suffer from low memory utilization, high allocation latency, or even allocation failure. Unlike those, this approach guarantees high memory utilization, low allocation latency, and success of allocations by utilizing a discardable pages based scheme. A summarized comparison between existing contiguous memory allocators and GCMA is shown in Table 1.1. Note that we are describing compaction disabled version of Buddy in this table.

Further, this paper demonstrates the effect of this allocator for system performance of the memory intensive workloads by adopting the allocator to the transparent huge page (THP) system. This replacement significantly improves system performance when the memory is highly fragmented. This result comes from the fact that traditional THP allocator fails to allocate contiguous memory for THP and falls back to normal page while our contiguous memory allocator guarantees the success to allocate. As a result, our allocator based system can provide THP benefit even in case of fragmentation.

1.2.2 An Optimization for High Scalability

The second approach introduces a new synchronization mechanism for read-mostly situation, which is based on a combination of the read-copy update (RCU) and the hardware transactional memory (HTM).

RCU achieves almost ideal performance and scalability for read-mostly situation, especially in case of in-kernel implementation. Nevertheless, as a cost of the almost ideal performance, RCU updates need to do more expensive and complex synchronization tasks. Also, RCU forces users to synchronize concurrent updates on their own instead of providing RCU-centric synchronization primitives or standardized principles to users. This is intended design of RCU that encourages users to choose best synchronization primitives for their specific case. That said, due to the bad reputation of the parallel programming, choosing or implementing synchronization is considered hard to many programmers. As a result, many RCU users are taking a global locking, which provides only poor performance and scalability for update threads.

To mitigate this situation, various RCU extensions providing higher performance and easier interface utilizing software- or hardware-transactional memory has proposed. However, those extensions had evaluated with only small or uniform memory access (UMA) system. Read-log-update (RLU) and RCU-HTM is two such state-of-the-arts extensions. For the reason, we evaluated the performances of these two extensions on a huge NUMA system. From the evaluation, the extensions revealed their high overhead that comes from their NUMA oblivious.

With extensive evaluations and experiments, we analyzed the reason of the overheads and found principles for design of a NUMA-aware RCU extension. The principles are:

1. Do fine-grained synchronization.

2. Use pure RCU read mechanism.

	RCU	RLU	RCU-HTM	RCX
Principle 1	X	O	O	O
Principle 2	O	X	O	O
Principle 3	X	X	O	O
Principle 4	N/A	N/A	X	O
Principle 5	N/A	N/A	X	O

Table 1.2 Summary of the principles adoption for each RCU extensions.

3. Use HTM.

4. Restrict HTM usage with multiple NUMA nodes.

5. Isolate the working set of HTM from readers.

Based on these principles, we designed our HTM-based NUMA-aware RCU extension. The approach combines HTM and traditional locking while keeping the principles and uses the combination as a synchronization primitives for RCU updates. We call this extension RCX. Adoption of the principles for each extension is summarized in Table 1.2.

Our micro-benchmark shows that the mechanism outperforms other state-of-the-art synchronization mechanisms based on RCU. We further adopts the synchronization mechanism to solve a well-known scalability bottleneck problem in Linux virtual memory management system to demonstrate its usefulness for the memory intensive workloads.

1.3 Dissertation Structure

Following chapter 2 describes related background, design, implementation, and evaluation of the guaranteed contiguous memory allocator. Chapter 3 describes and eval-

uates the RCU-based synchronization mechanism that is combined with HTM in NUMA-aware fashion. Finally, chapter 4 provides the conclusions.

Chapter 2

Guaranteed Transparent Huge Pages Allocations

2.1 Introduction

Recent trends in computing systems have greatly increased the importance of physically contiguous memory allocation. In particular, there has been increasing demand placed on small embedded systems, such as the Internet of Things (IoT) paradigm gaining wide market acceptance and smartphones spreading rapidly in emerging markets [5]. These small embedded systems are usually equipped with devices like video codecs or cameras which require large data buffers. These buffers can be scattered in system memory if additional hardware support is available. However, for many low-end systems, additional hardware is not affordable. As a result, the buffers should reside in physically contiguous memory.

Another important trend is the increase in the size of working set, especially in high-end server systems that are utilized for big data, cloud, or machine learning. Because large workloads tend to have low locality and high memory consumption [1],

these systems tend to face an increased number of translation lookaside buffer (TLB) misses. Therefore, modern high-end server systems tend to require petabytes of RAM. To efficiently manage this increase in capacity, Intel has recently attempted to increase the number of page table levels for x86_64 systems from four to five [2]. However, more page table levels inflates page table walk costs, which becomes an even greater problem in virtual machine environments that use extended page tables [6]. The combination of the increase in both TLB misses and page table walk costs can severely degrade performance. A well-known solution to this problem is the use of huge pages, which reduces TLB misses by enlarging the reach of the TLB. The allocation of a single huge page requires physically contiguous memory that is more than two orders of magnitude larger than that of a single normal page. Indeed, the success of contiguous memory allocation is essential for the performance of both small and high-end server systems. However, due to the inevitability of memory fragmentation, it is difficult to guarantee effective contiguous memory allocation.

A traditional solution for this problem is the memory reservation technique. It reserves a sufficient amount of contiguous memory early in the booting stage while the system is still in a less fragmented state. It then treats the area as a dedicated space exclusively for contiguous memory allocation, guaranteeing fast and successful results. However, memory reservation could lead to a depletion of memory resources if the demand for contiguous memory allocation is so low that the reserved area is not fully utilized. Some systems use a solution based on additional hardware, such as scatter/Gather direct memory access (DMA) and input-output memory management units (IOMMUs). These solve the problem by helping devices to access discontiguous memory as if it were contiguous, like the memory management unit (MMU) does for the CPU. However, the cost of additional hardware is not feasible for many low-end systems (e.g., Raspberry Pi) and the hardware only provides the illusion of contiguous allocation.

The well-known buddy memory allocation technique also supports contiguous memory allocation, though it has many limitations. It only supports allocations of 2^n contiguous pages because of its adjacent buddy-block based scheme. n is a natural number that possesses an upper bound. The Linux kernel for an `x86_64` system has a limit of only 10, which means it cannot allocate contiguous memory that is larger than 4 MB. Furthermore, the buddy system is susceptible to fragmentation.

The Linux kernel has developed a subsystem to deal with this problem, known as the contiguous memory allocator (CMA) [7]. It basically follows the memory reservation technique and increases memory utilization by allowing the reserved area to be used by other *movable pages*. If the areas used by *movable pages* are required for contiguous memory allocation, the CMA just migrates the *movable pages* out of the reserved area and lets contiguous memory allocation use the newly freed area. However, the CMA is not widely used in many real-world systems because of a number of serious limitations. The CMA typically exhibits high latency in allocation because it is required to wait for the completion of the migration of the movable pages. Furthermore, memory migration is inherently slow because pages are copied. It can even fail occasionally if the movable pages are not actually movable. Despite their name, movable pages cannot be moved if threads are holding them in place. Figure 2.1 shows the cumulative distribution function (CDF) of the latency for a photo taken on Raspberry Pi 2 [8] under stress from a real-world memory intensive task. "Reservation" indicates a system that uses the memory reservation technique to allocate a buffer for the camera, while "CMA" is a modified system that uses a CMA instead. The maximum latency for the photo using the CMA is about 9.8 seconds, while it is only 1.6 seconds using the reservation technique. Normally, 9.8 seconds is too long for a photo to be taken, even in the worst-case scenario. As a result, most systems, including Raspberry Pi 2, use the memory reservation technique or additional hardware as a last resort, despite the high costs involved.

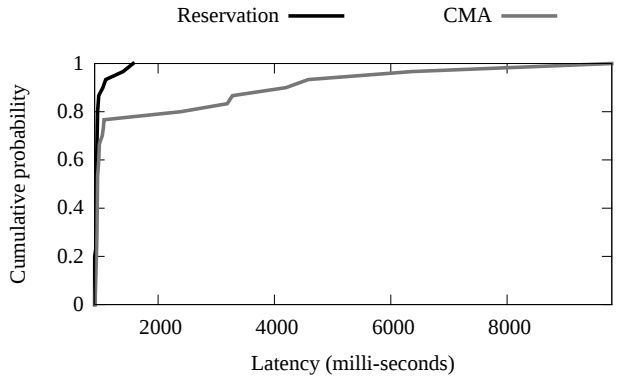


Figure 2.1 Cumulative distribution of the latency for a photo taken on Raspberry Pi 2 under memory stress.

In light of this, we propose a new subsystem for Linux called the guaranteed contiguous memory allocator (GCMA) [9]. It guarantees not only memory space efficiency, but also low latency and allocation success. The GCMA is similar to the CMA but with a key difference in the choice of additional clients. Unlike the CMA, which incorporates every page that is typically movable, the GCMA integrates only pages that are immediately discardable. On Raspberry Pi 2, the GCMA is 15 to 130 times faster and has a much more predictable response time for contiguous memory allocation without noticeable degradation of system performance. Furthermore, it incurs zero overhead for the system load in terms of the latency of taking a photo. On high-end server, the use of the GCMA for the allocation of huge pages improves the performance of a real-world huge working set by a speedup of up to 2.27x.

2.2 Background

2.2.1 Devices using DMA

Normal programs running on a CPU can ignore physical memory while accessing logical address space because the `MMU` transparently translates logical addresses into physical addresses from the perspective of the program. However, the `MMU` serves for only the CPU. In other words, devices that directly communicate with the memory cannot benefit from this translation process. When a device needs a large array in memory, but does not have the aid of the `MMU`, the array should be contiguous in the physical address space. One example of this is the processing buffer for a high-quality image. Such devices are prevalent these days, including cameras, ethernet cards, GPUs, and co-processors. Due to memory fragmentation, the allocation of large arrays has a high probability of failure, so devices requiring such an array cannot be reliably utilized.

Many systems avoid this problem by adding extra hardware units to act like the `MMU` for the devices. Scatter/Gather DMA and IOMMUs are well-known examples of these [7, 10]. Scatter/Gather DMA allows devices to scatter data to and gather data from distributed memory regions. IOMMUs provide devices with a translation from a logical address to a physical address, similar to the `MMU`. Nevertheless, additional hardware such as Scatter/Gather DMA and IOMMUs require extra costs and power consumption. For a system that is sensitive to sales price and/or energy consumption, these additional costs can be unaffordable. Even performance-critical high-end systems with the resources available to install additional hardware require contiguous memory. Lameter *et al*[10] recently evaluated remote direct memory access (RDMA) performance based on contiguous memory and Scatter/Gather DMA. Contiguous memory-based RDMA exhibited up to 50 % better performance than Scatter/Gather DMA.

2.2.2 Huge Pages

To minimize page table walk overhead, a TLB caches the results of previous address translations. However, if a working set is large enough to overflow the TLB, the benefits of using that TLB are lost. One well-known solution for TLB cache flooding is the use of huge pages. Each platform supports different huge page size. The `x86_64` system supports 2 MB and 1 GB huge pages in addition to the normal 4 KB pages. For the remainder of this paper, the term huge page will refer to 2 MB pages if no other explicit description is given because 2 MB huge pages are more useful than 1 GB versions in most cases. Because a huge page is more than two orders of magnitude larger than a normal page, it increases TLB reach and thus reduces the number of TLB misses. Because modern workloads, such as those for big data, cloud, and machine learning, are characterized by low locality and significant memory use, the use of huge pages has become more important. In virtual machine environments that use extended page tables, which is a common setup in cloud computing environments, page table walk costs have become even more expensive. Recently, Intel looked to increase the number of page table levels from four to five for systems equipped with a huge memory size. Doing so also increases the cost of page table walks. These recent trends highlight the importance of huge page utilization.

Linux has two approaches to huge page support: `hugetlbfs` and transparent huge pages (THP). `Hugetlbfs` is a traditional approach that programs can use to explicitly request huge pages. The system reserves a large contiguous memory area and utilizes this area as a pool for huge pages. Though `hugetlbfs` has long proven its usefulness, programmers are required to make additional code modifications to benefit from huge pages. The reserved area can also lead to memory wastage if the huge pages do not fully utilize the reserved area.

In contrast, THP is a relatively new approach. As the name implies, it works trans-

parently so that programs can benefit from huge pages without additional changes. The system transforms normal pages into huge pages and vice versa automatically. This process is conducted via a page fault handler and the `khugepaged` kernel thread daemon. In the page fault handler, Linux decides whether the faulted memory region should use huge or normal pages. By default, Linux selects huge pages for every region as long as possible. The `khugepaged` process scans the virtual address space of each program in the background and promotes the scanned normal pages if possible. Because the overhead required for `khugepaged` scanning can reduce system performance, the administrator can configure how aggressively `khugepaged` should be run. Unlike `hugetlbfs`, which uses memory reservation for huge page allocation, THP allocates contiguous memory for a single huge page to the buddy system with each promotion. Because programs can benefit from huge pages without any effort, THP is turned on by default in many Linux distributions.

Nevertheless, THP has a number of limitations. One example of these is memory bloating. Because it uses larger-sized pages, internal fragmentation can occur [6], wasting system memory. Programs that experience internal fragmentation, including Redis and MongoDB, have even recommended that users turn off THP [11, 12]. Another problem is contiguous memory allocation failure. When THP fails to allocate contiguous memory for a huge page, it falls back to just using normal pages. Thus, the benefits of THP are reduced if huge page allocation fails frequently. In fact, when faced with page allocation failure, earlier versions of Linux utilized memory compaction followed by and attempt at reallocation to preserve the benefits of THP. However, because compaction severely increases the allocation latency of huge pages [6], more recent versions of Linux have been modified to just fall back on the use of normal pages [13].

In order to illustrate the effect of THP and fragmentation, we set up the TPC-H benchmark with MariaDB on a Xeon CPU E7-8870 v3 processor system and execute

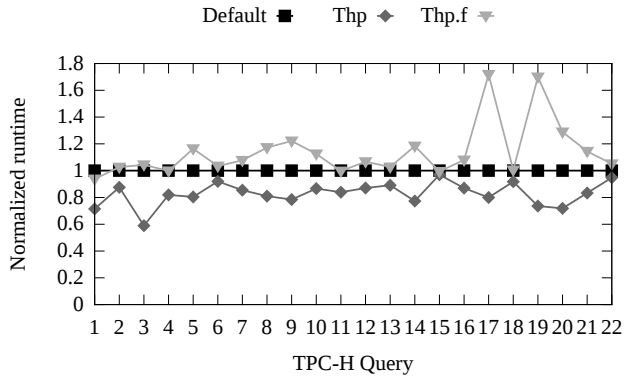


Figure 2.2 Negative effect of fragmentation on THP performance.

each query in TPC-H in a variety of environments, including THP-disabled (*Default*), THP-enabled (*Thp*), and THP-enabled with a fragmented memory (*Thp.f*). Thp.f fragments system memory so that only few huge page allocations would succeed. This type of fragmentation is not rare, but it is more common in older systems, including cloud infrastructure systems [14]. The workload is configured to use 4 GB of data by setting the *scale factor* to 4. Figure 2.2 shows the runtimes of each query, normalized by that of Default. THP improves the performance of every query by up to 40 %. However, with fragmentation, the performance of THP is worse overall than the default state. For example, Queries 17 and 19 for Thp.f have a slowdown of more than 60 % compared to Thp.

In short, huge pages are useful and THP can effectively contribute to their implementation. However, the benefits of THP are reduced in highly fragmented situations, which is a common situation in older systems.

We also note that many hardware-based approaches to the TLB misses exist. These approaches minimize TLB misses by translation speculation [15], coalescing multiple translations into single TLB entries [16], or allowing virtual-to-physical direct mapping [17].

2.2.3 Buddy Allocator

The buddy allocator system is the default memory allocator for the Linux kernel. It supports contiguous memory allocation, even though it is restricted in multiple ways. It splits memory into small blocks (usually of a normal page size) and manages these blocks. If two adjacent blocks are free and both are of the same size, the buddy system merges them together into a single, bigger block. As a result, every block in the system has a size of $2^n \times s$, where s is the size of the smallest block and n is an integer starting at zero. Clients are also restricted to requesting a memory size of $2^n \times s$. When an allocation request is received, the buddy system searches for a free block in accordance with the requested size. If the search is successful, it returns the found block. Otherwise, if a free block cannot be found, it searches for a free block that is larger than the requested size. If a larger free block is found, it continues to split the block in half until the requested free block size is available and then returns it. However, if the search for a larger free block fails, the allocation also fails.

Due to the merging and splitting of overhead, the buddy system limits the maximum value of n. In `x86_64` architecture, it is only 10 by default and s is 4 KB. In other words, in the default `x86_64` environment, the buddy system cannot allocate contiguous memory that is larger than 4 MB. The buddy system manages lists of free blocks for each n, so the allocation conducted by the buddy system is completed quickly and consistently for normal cases with the longest allocation time proportional to n. However, this can be more complicated in the real world in terms of compaction and reclamation. Nevertheless, the limitations of the buddy system in terms of contiguous memory allocation are obvious: it cannot support contiguous memory of sizes outside $2^n \times s$ and it is unreliable when it comes to fragmentation.

2.2.4 Memory Reservation

The memory reservation technique is the oldest and most widely used approach to contiguous memory allocation. It reserves a large portion of system memory in the early stages of booting, when there is only minor fragmentation. This reserved region is dedicated to contiguous memory allocation.

However, this approach can waste a significant amount of memory space. As briefly described for `hugetlbfs` in Section 2.2.2, an incorrect prediction of contiguous memory requirements can waste the memory inside the reserved area, when other processes require free memory. Despite this limitation, memory reservation remains the only approach that guarantees the success and low latency of contiguous memory allocation, which is why real-world systems continue to employ this solution as a last resort despite its limitations.

2.2.5 Contiguous Memory Allocator

As described above, both the buddy system and memory reservation have limitations. To overcome these limitations, the Linux kernel provides its own software solution for contiguous memory allocation, the contiguous memory allocator (CMA) [7]. The CMA, which is based on the memory reservation applies the basic concept of virtual memory, in which pages in the virtual memory can be mapped to any other place in the physical memory. As with the reservation technique, it reserves a large area of contiguous memory in the early stages of booting. However, while the reservation technique utilizes the reserved area only for contiguous memory allocation, the CMA accommodates not only contiguous memory allocation but also movable pages. If the movable pages interfere with contiguous memory allocation, the CMA just migrates them out of the reserved area to create a free area for allocation.

Because most of the pages in virtual memory are movable, the CMA approach can

guarantee high memory utilization. Because only movable pages are accommodated, migrating them out needs to be carried out in a similarly simple way. Unfortunately, however, real-world CMA typically exhibits unacceptably poor results. Its latency is typically high and the output is unpredictable, and allocation occasionally fails. The main reason for these problems is the high cost of page migration. Page migration is a rather complex process. First, the content of the page should be copied, and mapping between the logical address and the physical address should be re-arranged. Second, there could be any number of threads holding the page in the kernel's context, preventing it from being migrated until the threads release the page voluntarily. Waiting an arbitrarily long time is not reasonable, thus the CMA occasionally fails rather than waiting. As a consequence, the CMA guarantees neither low latency nor success.

2.3 Guaranteed CMA

The concept behind a guaranteed CMA (GCMA) can be described using a simple metaphor of rental rooms with different classes of client:

1. Rooms are exclusively owned by a manager.
2. The manager rents the rooms out to clients. Each client is classified as primary or secondary-class.
3. The manager reclaims the rooms of secondary-class clients if primary class clients request the rooms.

There are a number of approaches based on this concept that are not specific to the GCMA but which address general allocation problems. Even CMA is adopting it. The mechanism behind the CMA can be fully explained by subsituting the rooms

of the metaphor with the reserved area, primary-class clients with contiguous memory allocation, and secondary-class clients with movable pages. This means that the basic idea behind the CMA is no different to the GCMA or other successful resource allocators. The problem with the CMA lies in the selection of secondary-class clients, namely movable pages. As explained in Section 2.2.5, movable pages can require a long time to be moved or may even be unable to be moved. CMA designers assumed that movable pages would be sufficient for the rapid clearing of memory, but this has not been borne out in reality. As mentioned above, the basic idea behind the GCMA is similar to that of the CMA. The main difference is that the GCMA only chooses secondary-class clients that are able to be moved quickly enough.

2.3.1 Secondary Class Clients of GCMA

In the CMA process, movable pages cannot be immediately freed for contiguous memory allocation because their data needs to be preserved; this migration of movable pages also requires an arbitrarily long time. Therefore, appropriate secondary-class clients should satisfy the following three requirements: (1) they should be able to be freed at an affordable cost, (2) they should be heavily used enough to utilize the reserved area, (3) they should be out of kernel scope to avoid other threads pinning them down.

When a file is accessed, its content is loaded from the storage to the *page cache* [18] in the memory to increase the speed of subsequent access attempts. Writing to the file is cached in the memory and then flushed to storage at some later point. A page is classified as dirty if its content has changed but it has not yet been flushed back to the storage, and clean otherwise. If the page cache is full, it evicts pages that are not expected to be used in the near future. If the pages chosen for eviction are dirty, their content should be flushed before eviction, whereas clean pages can be evicted immediately. Because clean pages could be accessed again in the near future if the

page cache eviction logic is incorrect, the Linux kernel provides an interface for its cache, called Cleancache [19]. However, the implementation of cleancache backend is not provided by Linux. An implementation of the cache can use any memory other than system memory to accommodate the evicted clean pages.

The properties of the pages accommodated in the cache are a perfect match with the requirements of GCMA secondary-class clients. Because the pages are clean, immediately discarding them does not lead to any loss of information. Workloads are optimized to reduce writes because, in general, writes in major storage devices generate a high overhead. Thus, the number of clean pages will be large enough to utilize the area reserved by the GCMA. Most importantly, other kernel threads are not able to hold the pages because they have already been evicted from the kernel scope. Nonetheless, Cleancache utilizes file-backed pages only. If the system regularly uses anonymous pages because file access is rare, the reserved memory in the GCMA would remain in an idle state.

When the size of the used virtual memory exceeds that of the available physical memory, the kernel swaps out anonymous pages which are regarded as unlikely to be used soon. Because the estimation for future usage of the pages may be incorrect, the Linux kernel provides an interface that gathers together the pages to be swapped out, referred to as Frontswap [19]. Similar to Cleancache, the Linux kernel provides a Frontswap interface only; backend implementation is charged to users. One difference with Cleancache is that Frontswap can selectively cache hooked pages. For instance, if the Frontswap backend thinks that a given page will never be used again, it is able to refuse to accommodate it.

The properties of Frontswap mean that the pages cached into it can be candidates of the secondary-class client for the GCMA. The pages are out of the kernel's scope because, from the kernel's perspective, they have been swapped out. They are sufficient to cover the GCMA's reserved area because the pages cover almost every

anonymous page. However, cached Frontswap pages should be swapped out to a swap device before eviction. As a result, the allocation latency of the GCMA can be slowed by the speed of the swap device. To avoid this problem, we restrict the Frontswap backend to operating in write-through mode. In write-through mode, the pages given to Frontswap are also swapped out to the backing swap device at the same time. Because the content is always on the swap device, the GCMA can just discard the pages immediately for contiguous memory allocation. However, write-through can incur high overhead. The system can be slowed down if the swap device is extremely slow for the write, and the increase in the number of writes can reduce the lifetime of the backing swap device if it is based on flash memory. That said, administrators can configure their systems for these cases. This optimization is discussed in Section 2.3.2. In short, because the pages that are accommodated in write-through mode Frontswap are no longer required by the kernel and their content has already been written to the backing swap device, the pages can also be candidate secondary-class clients for the GCMA.

The Cleancache and Frontswap interfaces have proven effective in their wide use by numerous users, including OpenSUSE, Oracle, and Xen [19, 20]. For this reason, the GCMA selects the pages accommodated in Cleancache and Frontswap backends as its secondary-class clients. As a consequence, the GCMA can be considered transcendent memory [20] that utilizes a contiguous memory allocator as its primary-class client and final chance caches for evicting clean pages and swapping pages as its secondary-class clients. Figure 2.3 shows the workflow of a GCMA in a simplified form. The GCMA reserves system memory in the early stages of booting and handles contiguous memory allocation using the reserved area. The remaining system memory is used for normal memory allocation. When the space available in the system memory becomes too small, the system finds pages that are not expected to be used soon and evicts them. The GCMA receives anonymous pages and clean pages among these pages to

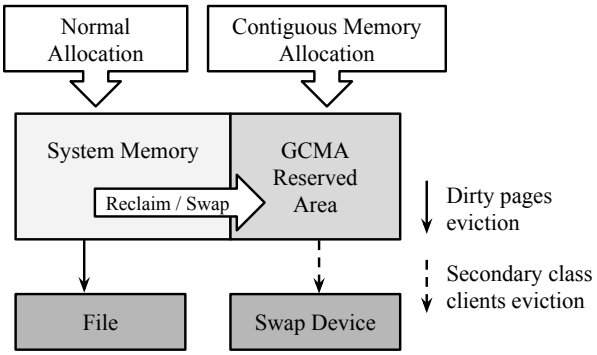


Figure 2.3 GCMA workflow.

be evicted, and keeps them in the reserved area. If contiguous memory allocation requires pages that are used for evicted pages, it reclaims the pages by discarding them immediately. If the system requests an evicted page again, it gives the page back if it still exists in the reserved area. Otherwise, if the page is reclaimed for contiguous memory allocation, it just returns a search failure result. In this case, the system will load the page from the file or from the swap device again.

2.3.2 Limitations and Optimizations

Although the secondary-class clients identified by the GCMA are effective and lightweight enough to utilize the entire system memory and guarantee low latency and the success of contiguous memory allocation, the GCMA can have an adverse effect on system performance compared to the CMA due to its design. The secondary client of CMA, movable pages, can be located in the reserved area with only a small overhead. In contrast, the accommodation of secondary-class clients for GCMA requires reclamation, which consumes processing power, and the rate of reclamation increases as the size of the reserved area becomes greater. However, our analysis indicates that this overhead is negligible. In addition, GCMA-based systems demonstrate slightly better performance than CMA-based systems because the secondary-class clients of

the former are not expected to be used in the near future.

Another limitation of the GCMA is the write-through overhead of swapped-out pages. As mentioned above, the GCMA restricts Frontswap to write-through mode in order to avoid slow write-backs during eviction for the primary client. In write-through mode, the content of the pages being swapped out will be written to the GCMA's reserved area and to the backing swap device at the same time. The GCMA can thus discard the pages without writing-back to the slower swap device. Although the use of write-through mode improves GCMA latency, system performance can be degraded because it consumes I/O resources. Furthermore, if the swap device is a write-count restricted device, such as a flash disk, the GCMA can shorten its lifetime. Nevertheless, this can be minimized by configuring the swap device with fast and write-immune storage. We suggest that GCMA users construct a swap device with compressed memory block device[21, 22] to enhance write-through speed, swapping performance, and memory utilization. Specifically, we recommend Zram[22] as a swap device, an official recommendation from Google [23] for Android devices with low memory resources. By optimizing the GCMA in this manner, we found that the GCMA suffered no performance degradation.

2.4 Implementation

The GCMA is implemented as a small Linux kernel module. In total, our implementation uses only about 1,300 lines of code for the main implementation and only about 200 lines of code for changes to the CMA. It is available as free/open source software under the *GPL v3* license at `https://github.com/sjp38/linux.gcma`.

The overall architecture of a system running a GCMA is depicted in Figure 2.4. The GCMA shares the CMA interface in order to be compatible with legacy programs. Client programs can select the CMA backend or the GCMA backend. The

system administrator can set the default backend as CMA or GCMA by simply turning on a single kernel option. There are three main components in the figure: *CMA*, *GCMA*, and *DMEM*. *CMA* is the default contiguous memory allocator of the Linux kernel. *GCMA* is the component for the contiguous memory allocation of the GCMA. For more efficient and flexible architecture, the GCMA only implements functions for its primary-class client, contiguous memory allocation. Instead of implementing and confining the operation of the reserved area for secondary-class clients in GCMA, it only defines the requirements and the interface for the operation and lets third-party developers implement theirs. Due to this separation, the GCMA can be configured with many different secondary-class client implementations, including our default selection, Cleancache and Frontswap pages. *DMEM* is our implementation of secondary-class client operations. As mentioned above, it utilizes the reserved memory based on Cleancache and Frontswap pages. For secondary-class clients, *DMEM* requests pages in the reserved area, while the GCMA can order DMEM to discard the pages for primary-class clients.

In summary, the contiguous memory allocating program uses the CMA's interface. Its backend can be CMA or GCMA. The CMA and GCMA use their own logic to find contiguous pages and allocate them to the reserved area. For system memory shortages, the system starts swapping anonymous pages out and/or evicting the page cache. The Frontswap and Cleancache interfaces receive the pages being swapped out and evicted. The interfaces are connected to DMEM. DMEM accommodates content from the swapped out or evicted clean pages inside the pages received from the GCMA. The content will be given back to the system if the swapped out or evicted clean pages are accessed again. After that, swapping and eviction are completed and the system can reallocate the swapped out or evicted pages to other programs.

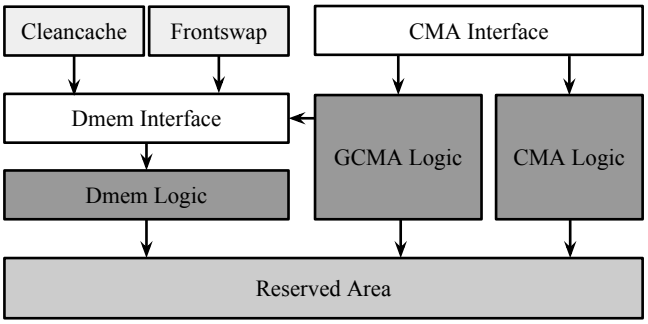


Figure 2.4 Overall GCMA architecture.

2.4.1 Contiguous Memory Allocation

The GCMA uses two bitmaps, one for the primary-class clients and the other for the secondary-class clients, to track the allocation state of each page in the reserved area. Contiguous memory allocation is processed as follows. First, candidate pages for currently issued contiguous memory allocation are found using the primary-class bitmap and the pages are checked to see whether they are allocated to secondary-class clients using the other bitmap. If so, the content of the pages for the secondary-class clients is discarded, marking them as free in the secondary-class bitmap. The pages are then marked as allocated in the primary-class bitmap and are then returned. The allocation process for a secondary-class client is conducted as follows. The GCMA searches for free pages from the primary-class bitmap, then it checks whether it is also free in the secondary-class bitmap. If this condition is satisfied, it marks the pages as allocated in the secondary-class bitmap and returns the page. Otherwise, the allocation fails. The use of the bitmaps means that the linear time complexity increases in proportion to the size of the reserved memory, though many forms of optimization are possible. We use the unoptimized implementation because the CMA also uses bitmaps in a similar manner.

2.4.2 DMEM: Discardable Memory

DMEM utilizes secondary-class clients in the reserved area by cooperating with the GCMA. The separation of the secondary-class client utilization logic from the GCMA allows alternative forms of implementation to be easily configurable with the GCMA. Each implementation may follow different policies as long as it follows the rules set down by the GCMA, i.e., it should support the rapid eviction of its pages from the reserved area. DMEM is just one possible example of such an implementation, though it is the only one at the time of writing. As its name indicates, DMEM is not limited to Cleancache and Frontswap. It can be expanded to accommodate other types of discardable memory in the reserved area.

DMEM is implemented as key-value storage for discardable objects. The GCMA system creates two instances of DMEM and assigns one to the Cleancache backend, and the other to the Frontswap backend. One crucial property of DMEM is that a privileged DMEM user can request a specific item be discarded at any time. This means that items that were stored before can be discarded in the interim, meaning that a subsequent search for that item can fail.

Because the number of items that are stored in DMEM can be fairly large, DMEM uses hash tables to manage these stored items with a reasonable latency. Each bucket is constructed as a red-black tree to scale well for a large number of items.

2.5 Guaranteed THP

THP in Linux uses the buddy system instead of a CMA for its 2 MB huge page allocations. This decision is reasonable because the buddy system provides contiguous memory allocation of up to 4 MB and is highly optimized for latency. On the other hand, CMA has been designed for larger, various sized, and less frequent contiguous memory allocation requests and is notorious for its high latency. However, as described

in Section 2.2.2, the buddy system fails to allocate huge pages in the presence of high memory fragmentation. In this situation, THP falls back to using normal pages and thus loses the benefit of using huge pages. Though using the CMA instead of the buddy system in this case would increase the allocation success rate, one drawback is the CMA's high latency. Because the GCMA provides successful low latency contiguous memory allocation, utilizing it for huge page allocation will solve the problem.

Despite this, this substitution is not a simple one for the following two reasons. First, the allocation speed of the GCMA can be slower than that of the buddy system in many cases, though it is much faster than that of the CMA. The time complexity of GCMA allocation is $O(n)$, where n is the size of the reserved area. The time complexity comes from its bitmap scanning. In comparison, the buddy system maintains a free page list so that $O(1)$ allocation latency is available in normal cases. As the policy and implementation of the GCMA are separated, it can be implemented in that way to allow a free list or to use other algorithms to achieve low latency. However, this type of optimization is beyond the scope of this paper. Secondly, it is questionable how much space should be reserved by the GCMA in order to accommodate huge pages. Because THP attempts to utilize huge pages as much as possible, an arbitrarily large number of huge pages can be requested to be allocated. However, reserving an arbitrarily large area for the GCMA will significantly shrink the system memory area available for normal pages.

To this end, we concluded that it is appropriate to configure the GCMA to act as the primary fallback for the THP allocation failure path, utilized before the fallback to the use of normal pages. If the system memory is not fragmented, THP will use the buddy system as before with its highly optimized low latency. On the other hand, if the system memory is highly fragmented, THP will fail to allocate huge pages using the buddy system, but it will attempt the allocation again with the GCMA, which will be successful in most cases. The higher latency of the GCMA in contrast

to the buddy system can be overcome by ensuring the benefits of the use of huge pages. We have implemented a prototype for this idea, called the Guaranteed THP, by modifying THP in Linux v4.10, which contains a recent change designed to improve THP allocation latency [13].

2.6 Evaluation

As described in Section 2.2, contiguous memory allocation is important in various environments including both low-end mobile systems and high-end server systems. Hence, we evaluate the GCMA in various kinds of environment.

2.6.1 Evaluation on a Mobile System

We use a Raspberry Pi 2 Model B single-board computer [8], which is one of the most popular low-end minicomputers in the world as our evaluation environment. The specifications for this system are given in Table 2.1.

Component	Specification
Processors	900 MHz quad-core ARM Cortex-A7
Memory	1 GB LPDDR2 SDRAM
Storage	Class 10 SanDisk 16 GB microSD card

Table 2.1 Raspberry Pi 2 Model B specifications.

The system configuration used in the evaluations, is presented in Table 2.2. Rather than the vanilla kernel, we use the forked kernel optimized for Raspberry Pi and provided by the Raspberry Pi development team on Github. The kernel we selected is based on `Linux v3.18.11`. We represent the kernel as `Linux rpi-v3.18.11`. Raspberry Pi has used the memory reservaton technique for contiguous memory allocation from the very beginning since it was introduced. After Linux implemented the CMA,

Name	Specification
Baseline	Linux rpi-v3.18.11 + 100 MB swap + 256 MB reserved area
CMA	Linux rpi-v3.18.11 + 100 MB swap + 256 MB CMA area
GCMA	Linux rpi-v3.18.11 + 100 MB Zram swap + 256 MB GCMA area

Table 2.2 Configurations for evaluations.

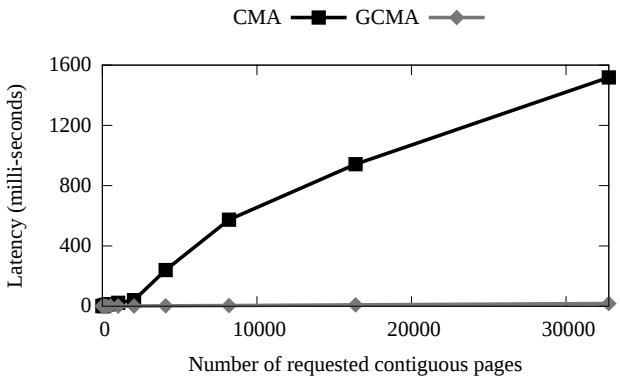


Figure 2.5 Averaged allocation latency.

Raspberry Pi started to support the CMA from November 2012 [24]. However, the Raspberry Pi development team has found problems with the CMA and has decided to only support it unofficially [25]. As a result, the default configuration for Raspberry Pi still uses memory reservation.

Our evaluation focuses on three factors: (1) the latency of contiguous memory allocation, (2) the latency of a real workload, in this case the Raspberry Pi Camera [26], and (3) system performance.

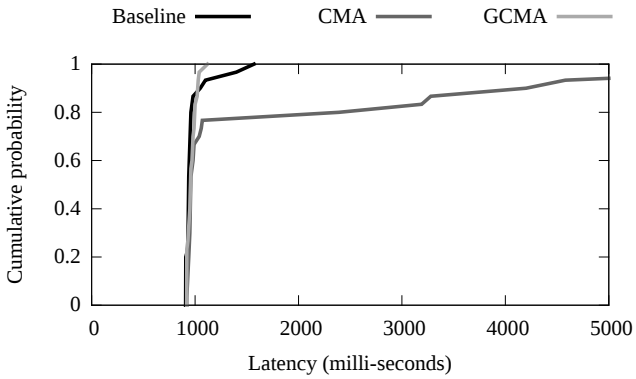
Latency of Contiguous Memory Allocation

To compare the contiguous memory allocation latency of the CMA and GCMA, we issue contiguous memory requests of varying allocation sizes. The first request is for 64 pages (256 KB), with the size doubling for subsequent requests until the last request, which is for 32,768 pages (128 MB). There is a 2 second interval between each allocation to minimize side effects. We repeat this 30 times for both the CMA and GCMA configurations.

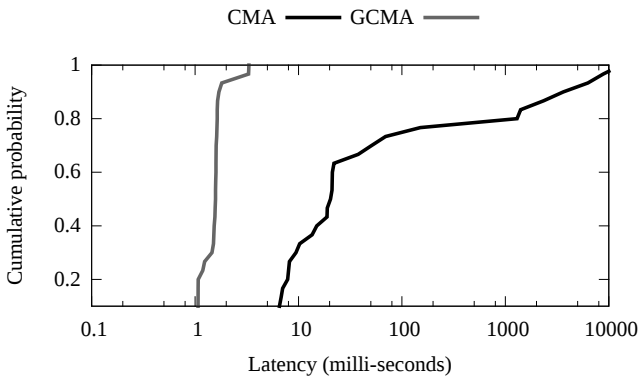
The average latency of the CMA and GCMA is shown in Figure 2.5. The GCMA exhibited `14.89x` to `129.41x` lower latency compared to CMA. In addition, the CMA failed once for the 32,768 page allocation while GCMA did not fail, even though the workload was run with no background tasks. Even without any background tasks, the CMA still has secondary-class client pages that need to be moved out of the reserved area because any movable pages can be located in the reserved area. In this case, moving pages out consumes a significant amount of time and can lead to failure in the worst-case scenario, as seen in the 32,768 page allocation. On the other hand, because only pages to be evicted can be located inside the reserved area for the GCMA, the GCMA does not need to discard any pages from the reserved area unless memory intensive background workloads require eviction and/or swapping. Moreover, the CMA code is much larger and slower than that of the GCMA because the CMA has to consider the complications associated with movable pages and migration while the GCMA only needs to consider *DMEM*. That is why the GCMA shows much lower latency than the CMA, even without background tasks.

Raspberry Pi Camera Latency

In this section, we evaluate the latency predictability of the CMA and GCMA using a real application, the Raspberry Pi Camera. When a photo is taken, Raspberry Pi 2



(a) Camera latency



(b) Contiguous memory allocation

Figure 2.6 CDF of camera / following memory allocation latencies under Blogbench.

usually allocates 64 contiguous pages 25 times asynchronously using a kernel thread and keeps the allocated memory as long as possible [24] to use it again for subsequent photo. Therefore, only the first photo taken requires contiguous memory allocation. To streamline the evaluation, we configure the system to release the memory allocated for the camera as soon as possible for subsequent shots.

Without background tasks, every configuration demonstrates a camera latency of about 0.9 seconds. There is no significant difference between the latencies of the photos taken using the CMA and the GCMA because the latency is dominated by the camera application, not by contiguous memory allocation. To illustrate the latency in a realistic environment, we run the camera workload in conjunction with a background task. For this background task, we use the `Blogbench` benchmark [27]. This is a portable file system benchmark that simulates a real-world busy blog server. To minimize the effects of scheduling on latency, we set the priority of taking a photo to be higher than that of the background workload. We measure the latency of each photo taken with the Raspberry Pi Camera 30 times for each configuration at 10 second intervals between each photo to eliminate any effect from previous photos. To explicitly demonstrate the effect of contiguous memory allocation, we measure the contiguous memory allocation latency for each photo using the CMA and the GCMA.

The results are presented in Figure 2.6. The CMA demonstrates a much slower camera latency while the GCMA has a similar latency to that of the baseline using the memory reservation technique. At their slowest, the CMA requires about 9.80 seconds to take a single photo while the GCMA requires 1.04 seconds. The contiguous memory allocation latency also exhibits similar but more dramatic results. The latency of the CMA lies between 4 milliseconds and 10 seconds, while that of the GCMA lies between 750 microseconds and 3.32 milliseconds. This means that contiguous memory allocation using the CMA produces extremely high and unpredictable latency in a real-world situation. Based on these evaluation results, it is not surprising that the

CMA is not officially supported on Raspberry Pi [25].

Effect on System Performance

Finally, to illustrate the effect of the CMA and the GCMA on system performance, we measure system performance for every configuration using two benchmarks. First, we run a microbenchmark called lmbench [28] to demonstrate the performance of OS primitives under the CMA and the GCMA. We run the benchmark three times and average the results to minimize noise. The rows in Table 2.3 show the context switch latency, file read bandwidth, memory read bandwidth, and memory write bandwidth, respectively. The CMA and GCMA tend to demonstrate improved performance compared to the baseline owing to their better memory utilization though the difference is minor, being within the margin of error.

	Baseline	CMA	GCMA
lat_ctx(usec)	147.36	143.53	142.93
bw_file_rd(MB/s)	511.77	517.60	519.73
bw_mem_rd(MB/s)	1426.33	1438.33	1434.50
bw_mem_wr(MB/s)	696.50	701.25	699.97

Table 2.3 LMbench measurement results.

We next run Blogbench to illustrate performance when handling a real-world workload. We continuously take photos as a background task at 10 second intervals, as described in Section 2.6.1, to simulate realistic contiguous memory allocation stress.

Again, we run each configuration three times and average the results, which are shown in Figure 2.7. Results with the addition of the background task are represented by the `/cam` suffix, while the results without the background task do not have this suffix. The CMA and GCMA exhibit higher performance in every case, though the difference is not large. These performance gains stem from the greater memory space

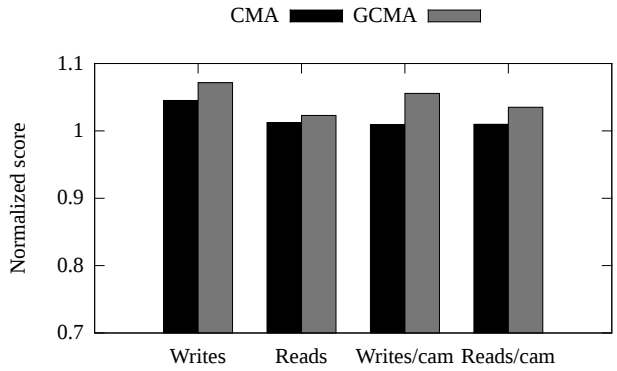


Figure 2.7 Blogbench performance of the CMA and GCMA.

that is rescued from the memory reservation technique. The GCMA has a slightly better performance than the CMA. The use of background contiguous memory allocation in particular creates less of a performance drop for the GCMA than for the CMA. In short, the GCMA is not only faster than the CMA but also more beneficial for system performance.

2.6.2 Evaluation on a Server System

The system used for the evaluation of the Guaranteed THP was a high-end server workload utilizing four Intel Xeon E7-8870 CPUs running at 2.10 GHz. The processor has a 46,080 KB last level cache and L2 TLB for 4 KB / 2 MB pages with 1,024 entries. The system is equipped with 600 GB of DDR4 memory and a 500 GB Intel SSD.

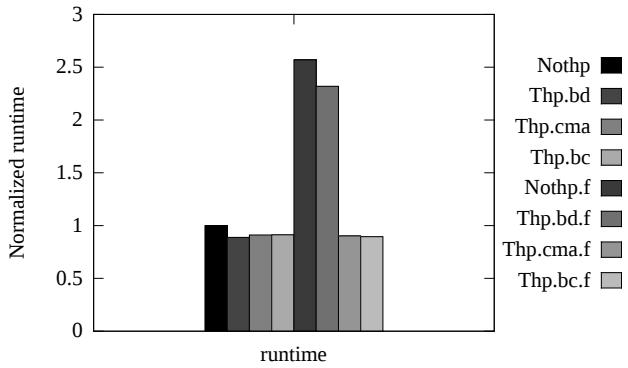
To illustrate the effect of huge pages for various situations and configurations, we use eight systems configured to use huge pages in different ways and in different situations. A brief outline of the variants is presented in Table 2.4. All of the variants are based on Linux v4.10, which contains improvements in THP latency [13]. *Nothp* is the default Linux system configured not to use THP while *Thp.bd* is similar to *Nothp*, but it is configured to use the Linux THP , which uses the buddy system for huge

page allocation. *Thp.cma* uses THP, but it uses a GCMA as a huge page allocator instead of the buddy system. *Thp.bc* is the implementation that was mentioned in Section 2.5. It is comparable with *Thp.bd* except that it uses the GCMA as a fallback for the allocation failure of the buddy system. The other four systems have the same names as the systems described above but include the suffix "*.f*". Each of these are characterized by memory that is highly fragmented.

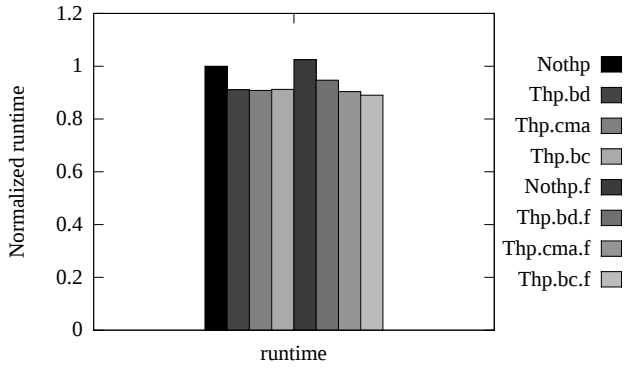
Name	THP	Allocator	Fragmentation
Nothp	N	N/A	N
Thp.bd	Y	Buddy system	N
Thp.cma	Y	GCMA	N
Thp.bc	Y	Buddy system with GCMA as a fallback	N
Nothp.f	N	N/A	Y
Thp.bd.f	Y	Buddy system	Y
Thp.cma.f	Y	GCMA	Y
Thp.bc.f	Y	Buddy system with GCMA as a fallback	Y

Table 2.4 Evaluated system configurations.

To conduct fragmentation, we use a simple program that is similar to that used in Kwon *et al*'s approach [6]. It allocates a large portion of the system memory and then frees random pages. It is manually configured so that the free space in the system memory exceeds the working set size of the workload to be evaluated but is insufficient to accommodate the entire working set as huge pages only.



(a) 429.mcf



(b) 471.omnetpp

Figure 2.8 SPEC CPU 2006 runtime with variants of THP.

Performance Using Real Workload with the Guaranteed THP

SPEC CPU 2006 [29] is a benchmark suite which contains various benchmarks that simulate real-world applications. Although SPEC CPU 2006 was primarily developed to test CPU performance, a few of its benchmarks are well known for their memory intensiveness [30]. Therefore, we choose two of these, 429.mcf and 471.omnetpp, to evaluate the Guaranteed THP for realistic workloads. 429.mcf handles single-depot vehicle scheduling in public mass transportation. Its specification explicitly requires 1,700 MB of memory for a 64-bit data model. When we consider recent trends in large memory workloads in high-end server, memory requirements are actually relatively small. However, Kwon *et al*[6] have reported that Linux THP can provide a 43% speedup compared to the benchmark in virtual machine environments. 471.omnetpp conducts a discrete event simulation of a large Ethernet network. It does not explicitly specify its memory requirements, but it is known to be memory intensive because of its high cycles per instruction (CPI) [30].

We run the benchmarks on the eight systems five times and average the runtime. Figure 2.8 displays the results.

Both Thp.cma and Thp.bc produce consistent speedups regardless of fragmentation. In contrast, both Nothp and Thp.bd experience slowdowns with fragmentation.

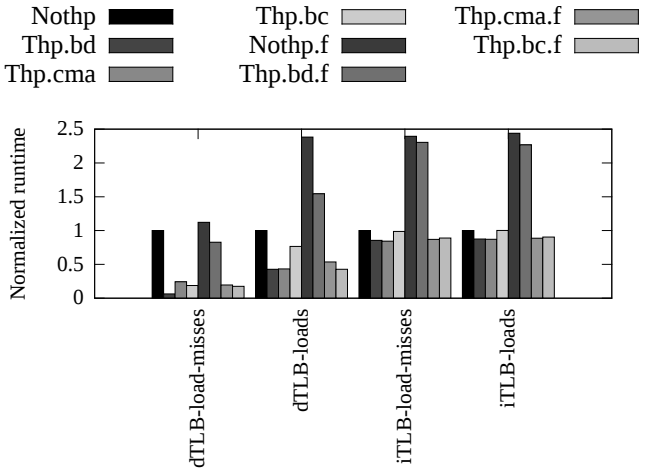
Although the trend is similar for both workloads, the extent of the impact is significantly differnt. In Figure 2.8(a), the Guaranteed THP (both Thp.cma.f and Thp.bc.f) produce a performance that was 2.56x higher than default THP under fragmentation. Note that the most significant contributor to this speedup is the physical memory contiguity sensitive of 429.mcf. The reason of this difference is illustrated by their different TLB event counts that shown in Figure 2.9). Performance of Nothp under fragmentation shows about 2.5x times lower performance compared to no fragmentation situation and the TLB event count shows the reason. This is because prefetched

TLB entries becomes useless under the fragmentation. In other words, the performance of 429.mcf is sensitive to the physical contiguity of memory access. Thp.bd.f utilizes as many huge pages as possible, but the number of utilized huge pages is much smaller than that of Thp.bd due to fragmentation. As a result, the physical contiguity of memory access is similar to Nothp.f, though the small number of huge pages enhances performance by reducing TLB misses and preserves the physical contiguity of memory access for the huge pages. In contrast, because Thp.cma.f and Thp.bc.f still succeed in allocating huge pages under fragmentation, they receive the full benefits of the reduction in TLB misses and the physical contiguity of memory access inside the huge pages. The performance gain from TLB misses will increase further as the working set size grows.

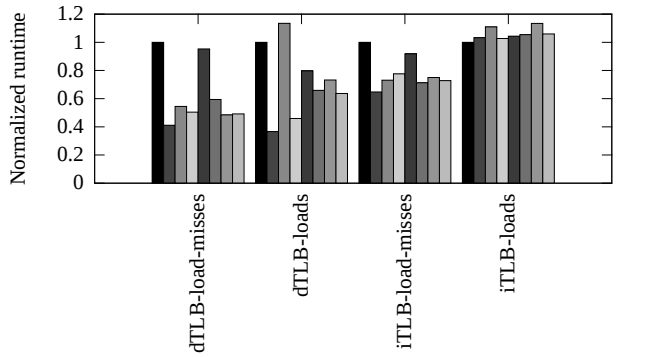
Figure 2.8(b) and 2.9(b) also show that the 471.omnetpp's modest performance changes comes from its insensitivity to the physical contiguity of memory access. The performance of 471.omnetpp does not change significantly in terms of the physical contiguity of memory access though it shows consistent performance change. However, the working set of 471.omnetpp is less than 200 MB in size. This small working set is a reasonable explanation for the small effect of the Guaranteed THP, with the performance consistent with a relatively small working set.

Finally, we evaluate the systems using the TPC-H benchmark [31] on MariaDB 10.2.8 [32]. This benchmark simulates a decision support system for a large web commerce. We installed MariaDB to use the default configuration for huge systems (`my-huge.cnf`) provided by the MariaDB source code, except for the following changes:

```
innodb_buffer_pool_size = 10240M
innodb_log_file_size = 256M
innodb_log_buffer_size = 16M
innodb_flush_log_at_trx_commit = 1
```



(a) 429.mcf



(b) 471.omnetpp

Figure 2.9 Numbers of SPEC CPU 2006 tlb events.

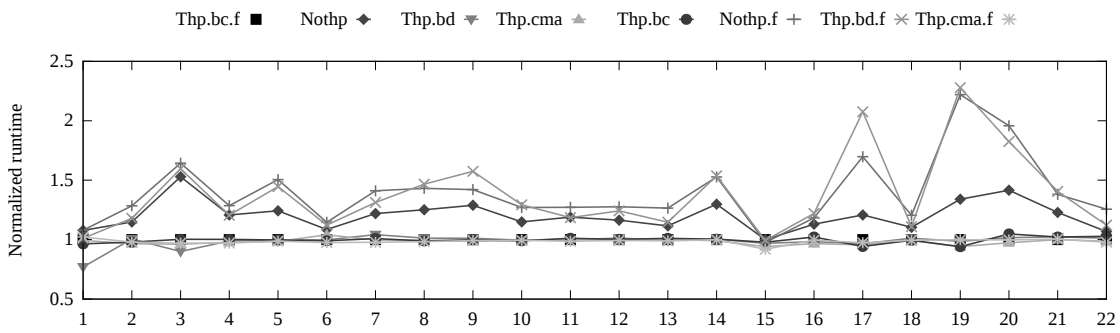


Figure 2.10 TPC-H strong test runtime on variants of THP.

The first three parameters force MariaDB to utilize a 10 GB buffer pool, a 256 MB log file, and a 16 MB log buffer. The last parameter forces it to flush the log with every transaction.

To evaluate the Guaranteed THP using the TPC-H benchmark, we first do a *load test* for TPC-H with a scale factor of 4; a 4 GB data set is loaded and its index is constructed on MariaDB. After that, we run a *power test* for TPC-H. The test sequentially executes and measures the runtime of 22 complex queries. As with previous evaluations, we repeat the test five times and average the results. Figure 2.10 presents the results of each TPC-H query for the eight systems. The x-axis of the figure shows the query number, while the y-axis shows the runtime for each query normalized by that of the Thp.bc.f system.

The overall behavioral trend of each query is similar to previous evaluations. The variants that benefit from huge pages without allocation failure, Thp.bd, Thp.cma, Thp.bc, Thp.cma.f, and Thp.bc.f, exhibit the best performance. The other systems that cannot benefit from huge pages, Nothp, Nothp.f, and Thp.bd.f, produce slowdowns. The overall trend means that the Guaranteed THP consistently supports system performance, as illustrated by the previous analyses. Performance gains from the GCMA vary for each query because they are dependent on the characteristics

of each query, but most exhibit a consistent speedup. Because Thp.bd.f is a common environment in high-end server systems, the comparison between Thp.bd.f and Thp.bc.f will allow the benefits of the Guaranteed THP to be fairly assessed. Two queries (17 and 19) produce a speedup of more than 2x, while four queries (3, 9, 14, and 20) show a speedup of 1.5x–2x. Query 19 produced the greatest speedup with 2.27x. Nonetheless, a few queries generated only a negligible speedup of less than 10%, including Queries 1 and 15.

One outlier result can be seen with Query 1 for Thp.bd. That is because Query 1 is the one which is run first. Therefore, the system has not warmed up yet, resulting in inconsistent behavior.

In short, our evaluation indicates that the Guaranteed THP consistently improves system performance in the presence of fragmentation, though the extent of the speedup is based on the workload characteristics. There was a speedup of more than 1.5x compared to the default Linux THP in the presence of fragmentation for 7 of the 24 workloads.

2.7 Related Work

Android is the number one platform on the smartphone market and is widely used for various embedded or IoT devices. The ION memory allocator [33, 34] is a memory pool manager for Android which is responsible for contiguous memory. Android device vendors can configure the ION to manage memory pools in various ways, including IOMMUs, Scatter/Gather DMA, CMA, or the memory reservation technique. However, if the device has neither Scatter/Gather DMA nor IOMMUs, it has no choice except for the CMA or memory reservation. Consequently, low-end smartphones still suffer from contiguous memory allocation, whereas high-end smartphones can avoid the problem using hardware support. Alternative platforms in the embedded market,

such as Tizen and Web OS, have attempted to use CMA [35] and to overcome its limitations. However, their solutions tend to be closed workarounds rather than a complete, open solution due to the closed manner in which industries and market actors operate.

Jeong *et al*[36, 37] proposed an approach for contiguous memory allocation and memory utilization in embedded systems. Their approach is somewhat comparable with the GCMA. They utilize the reserved area by renting the area to clean pages evicted from page caches. Nevertheless, this approach can be negatively affected by the intensive workload of anonymous pages and exclusively focused on embedded systems. The GCMA, on the other hand, is designed for both embedded and high-end systems.

Kwon *et al*[6] developed a memory management system for THP called Ingens that has been optimized to solve two issues: high tail latency arising from memory compaction for huge page allocation and memory bloat from the internal fragmentation of huge pages. It solves tail latency by deferring normal-to-huge page promotions and alleviates memory bloat by promoting only if the promoted huge page is expected to be heavily utilized. Since then, a similar approach for the enhancement of THP allocation latency [13] has been proposed and merged into the mainline Linux kernel. However, the deferring approach cannot completely take advantage of the benefits of huge pages in the presence of high memory fragmentation because almost every huge page allocation will fail, and the system will just fall back to using normal pages. The Guaranteed THP is designed and implemented on Linux v4.10, which includes Linux's mainline deferring approach and aims to guarantee the success of huge page allocation. The evaluation results from Section 2.6.2 show that the Guaranteed THP can further improve the performance of a system that applies the deferring approach. Because the mainline deferring approach is quite similar to that of Ingens, we expect that if Ingens utilizes the Guaranteed THP, a similar improvement will result.

Ingens's solution to memory bloat can also be easily combined with the Guaranteed THP because the Guaranteed THP does not affect the promotion decision logic.

2.8 Conclusion

Physically contiguous memory is important for various computing environments, including low-end embedded systems and high-end server systems. However, existing allocators all have a number of limitations. For example, Scatter/Gather DMA and IOMMU-like hardware-based solutions increase the price and power consumption, the buddy system is susceptible to fragmentation, the memory reservation approach reduces memory utilization, and CMA is too slow.

We have introduced the GCMA, a contiguous memory allocator that guarantees low latency, successful allocation, and reasonable memory space efficiency. Based on our assessment using Raspberry Pi 2, the GCMA has a 130x and 10x shorter latency for contiguous memory allocation and photo taken by the device's camera, respectively. The GCMA also improves system performance. The analysis of the use of the GCMA on a high-end server system demonstrated that the GCMA can enhance huge working set performance in the presence of fragmentation. With 24 different real-world workloads, it displays a steady gain in performance. Specifically, it obtained more than 50% for seven workloads, and an improvement of over 100% for three of those.

Chapter 3

A Scalable Virtual Address Space Protected by an HTM-based NUMA-aware RCU Extension

3.1 Introduction

RCU provides ideal read-side performance and scalability, further providing concurrent forward progress among readers and updaters [38], in contrast to traditional read-centric synchronization mechanisms such as reader-writer locks [39]. However, RCU imposes relatively heavy overhead on updates, and further defers to other synchronization mechanisms to synchronize between concurrent updates. This design choice allows RCU to work with whatever update-side synchronization mechanism is the best for the case at hand, or even to work with an ad hoc mechanism devised for a particular special case [40]. The downside of this tradeoff is lack of clear guidelines. As a result, many RCU users use simple synchronization mechanisms, such as global locking, which do not scale well, thus leading some to incorrectly believe that poor update-side scalability is inherent to RCU.

One response to this situation has been to develop widely used RCU-protected concurrent data structures, including hash tables [41, 42] and binary trees [43]. These data structures have been successfully adopted, for example by a distributed file system [44] and by an in-memory database [45]. That said, hash tables and search trees do not always suffice: Ad-hoc data structures are often required.

Another response has been to use techniques similar to software transactional memory (STM) such as read-log update (RLU) [46, 40], hardware transactional memory (HTM) [47, 48, 49, 50, 51], and HTM in conjunction with RCU [52]. In Sections 3.2.2 and 3.2.3 we show that these approaches suffer from a number of limitations, including scalability limitations due to NUMA-obliviousness and due to naïve fallback mechanisms for HTM failures.

To mitigate these problems, we introduce another synchronization mechanism combining HTM and RCU, along with an RCU extension called RCX that incorporates this mechanism. RCX's advantages stem from these properties: (1) HTM minimizes synchronization overhead [53], (2) Hierarchical synchronization optimizes for NUMA characteristics, and (3) Read-side overhead is minimized through use of standard RCU techniques. More specifically, RCX isolates HTM-based synchronization to threads in same NUMA node because otherwise HTM abort rates rise sharply. Updater threads within a given NUMA node elect a leader using HTM, and then elected leaders synchronize using traditional locking. Locking it scales well in RCX due to the limited number of contending threads. Reader threads use RCU, thus attaining RCU's traditional ideal performance and scalability. Furthermore, RCX isolates meta-data for update-side synchronization from readers to eliminate HTM aborts induced by conflicts between readers and update-side synchronization.

To evaluate our approach in detail, we implemented linked lists and hash tables protected by various RCU extensions, including state-of-the-art mechanisms and the RCX. In addition to illustrative micro-benchmarks, we applied RCX to a portion

of the Linux-kernel virtual memory system to obtain a realistic system-level RCX evaluation.

In short, this paper investigates state-of-the-art RCU extensions on NUMA systems (Section 3.2 and 3.3.1), introduces our update-side synchronization mechanism for RCU on NUMA systems (Section 3.3.2), and presents adoption of the mechanism to data structures and a virtual memory system (Section 3.3.3 and 3.4). A free/open-source implementation of our mechanism and related programs will be available.

3.2 Background and Related Work

Because power-consumption and heat-dissipation issues have limited CPU core clock frequencies [54], systems with hundreds of cores have been common, pushing parallel programming to the fore. However, Amdahl's Law dictates scalability limitations for real applications. Furthermore, use of expensive synchronization primitives [55, 53] often causes synchronization costs to increase with the number of CPUs [56, 57]. No single state-of-the-art concurrency control mechanism works best for all workloads [58]. Therefore, achieving high performance on multicore systems requires careful selection of concurrency control mechanisms.

3.2.1 Read-Copy Update

RCU protects shared data from concurrent threads based on two core concepts. **1) RCU read-side critical section:** Several parts of the code reading the data can be marked as being in the read-side critical section. RCU guarantees any read from the data in the section is safe. **2) Grace period:** RCU provides a primitive to wait unitl every *pre-existing* RCU read-side section is completed. The waiting time is called the *grace period*.

Example pseudo code for reads and updates for data protected by RCU in the

Linux kernel is shown in Listings 3.1 and 3.2, respectively. To read data items, a reader thread should notify RCU that it will read data by invoking `rcu_read_lock()` (line 1). After the notification, the thread can access the data safely. On weakly ordered architectures, such as DEC Alpha, careful use of additional memory barriers is necessary, although it is not required for systems providing strong ordering guarantees [59]. For simplicity, RCU provides `rcu_dereference()` (line 2), which embeds the necessary memory barriers based on the underlying architecture. Once the reads are complete, the thread should again notify the RCU by invoking `rcu_read_unlock()` (line 4). The read-side critical section should be short and have no sleepable operations in it, though there exist sleeping allowed RCU variants, such as SRCU [60].

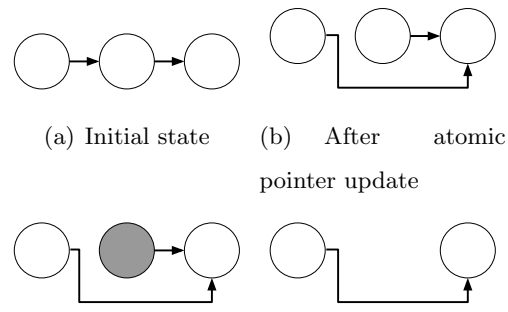
To insert a new data item into a linked list (lines 1–5 in Listing 3.2), a thread should fill a memory region for the item with proper values (line 3) and then atomically link the item via atomic pointer modification and necessary memory barriers (line 4) so that readers can reach the item. Figure 3.1 shows the memory state for the removal of a data item (lines 7–12 in Listing 3.2). White circles represent items that some readers can access, while gray circles represent items that no reader can access. Arrows means `->next` pointer. To remove a data item, a thread again uses the atomic pointer modification and necessary memory barriers (line 9). After the atomic modification (Figure 3.1(b)), only two kind of readers exist. 1) Readers started after the modification and 2) *pre-existing* readers. While the readers started after the modification cannot access the item, the pre-existing readers may fetched a reference to it before. Because the pre-existing readers can access the item again via the reference, the updater thread should not destruct or reuse the memory region for the item. Instead, it should wait until every pre-existing reader completes their critical section (line 10). After grace period (Figure 3.1(c)), no reader is able to access the item. Thus, the updater can destruct or reuse the region (line 11). This is called quiescent state-based reclamation (QSBR).

```
1 rcu_read_lock();
2 p = rcu_dereference(head);
3 do_something(p);
4 rcu_read_unlock();
```

Listing 3.1 RCU read critical section.

```
 1 void insert(node, prev, next)
 2 {
 3       node->next = next;
 4       rcu_assign_pointer(prev->next, node);
 5 }
 6
 7 void remove(node, prev, next)
 8 {
 9       rcu_assign_pointer(prev->next, next);
10       synchronize_rcu();
11       kfree(node);
12 }
```

Listing 3.2 RCU update code for a linked list.



(a) Initial state (b) After atomic pointer update

(c) After a grace period (d) After the destruction of the item

Figure 3.1 RCU-protected data item removal.

Another widely used synchronization mechanism for read-mostly situations is reader-writer locks [39] which provide mutual exclusion between readers and updaters. This can degrade the performance of read-mostly workloads because readers can wait for updaters. Contrarily, RCU readers never wait for anything. Further, with the Linux kernel implementation for common configurations, marking the entrance and exit of the read-side critical section has no cost at all because the marking functions are compiled to `noop`. That is, RCU readers do not wait for anything, do no synchronization at all but just read with required memory barriers, which is not required on strongly-ordered systems. In this way, RCU provides the almost ideal performance and scalability of reads, which no other synchronization mechanism can surpass.

That said, the code in Listing 3.2 cannot be executed by multiple concurrent threads without any synchronization because it can corrupt the data. RCU itself does not provide synchronization mechanisms for the update-side. Instead, it forces users to synchronize their concurrent updater threads by themselves. This is an intended design aspect to let clever users use or invent the best synchronization mechanism for their given environment. However, users commonly embrace unscalable but simple global locking due to the difficulty of parallel programming. As a result, RCU has been known to have poor performance and scalability for updates [46, 40].

3.2.2 Hardware Transactional Memory

HTM allows programs to group multiple memory accesses into atomic transactions, which in turn provides a simple program environment and high performance compared to that of software-based synchronization mechanisms. Although no every CPU vendor implements unlimited HTM on their products, several vendors are supplying HTM-enabled products, either in production [47, 48, 49] or as a research prototype [50]. Although HTM has many restrictions, it has been successfully adopted by several workloads [51, 52]. Siakavaras *et al.*[52] proposed a combination of RCU

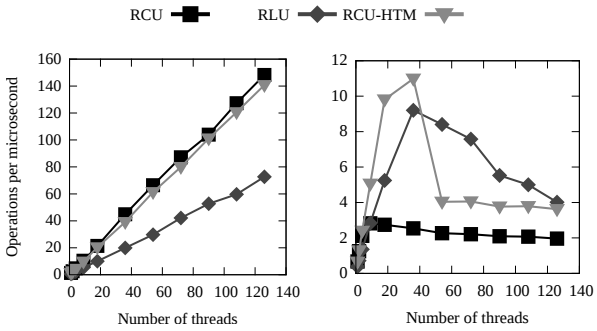
and HTM (RCU-HTM) that eliminates locking for updates by processing updates in HTM transactions. This paper uses Intel's HTM implementation [61].

Intel's HTM maintains a read data set and a write data set in the CPU cache and checks for data conflict between CPUs based on the cache coherency protocol. If a transaction is writing a data item that is read or written by other concurrent CPUs, HTM aborts and rolls back the intermediate changes of transactions involved in the conflict. Intel suggests HTM users use HTM in two ways, hardware lock elision (HLE) and restricted transactional memory (RTM). HLE replaces memory operations for lock acquisition and release with HTM-based operations and falls back again to software-based traditional locking if the transaction is aborted. HLE is easy to adopt because only the source code for the lock implementation needs modification. RTM is more analogous to traditional transaction systems. Users should explicitly use HTM transactions and provide fallback mechanisms for aborts. In short, compared to HLE, RTM is harder to adopt but is highly optimizable.

Although HTM incurs lower overhead and scales better compared to software-based synchronization mechanisms in common cases, it cannot entirely substitute software-based synchronizations. The careless use of HTM can result in insufficient or even worse performance because most implementation of HTM in real products have many restrictions [51], including transaction size limitations and unexpected abortions for various reasons.

3.2.3 Related Work

Ramadan *et al.* present TxLinux, which applies transactions to an early version of the Linux kernel [62]. Given that the point of their work was to transactionalize the Linux kernel as opposed to interacting with RCU, they took the pragmatic approach of aborting any transactions containing accesses conflicting with an RCU read-side critical section. This work did not involve a commercial HTM implementation and



(a) Read-only (b) 20 percent updates

Figure 3.2 Performance of RCU, RLU, and RCU-HTM on linked lists with 256 initial nodes.

did not address NUMA performance issues.

Howard *et al.* present an STM that is enhanced to accommodate RCU readers, which they call "relativistic readers" [63, 64]. They use SwissTM [65], which provides weak atomicity (permitting concurrent readers) and invisible uncommitted data (avoiding transaction aborts). They extended SwissTM to commit writes in program order and to respect ordering directives, including memory barriers and RCU grace-period-wait operations. As far as we know, this is the first work permitting concurrent forward progress among RCU readers and TM writers. However it does not involve HTM, and nor does it address NUMA performance issues.

McKenney describes a number of additional strategies for interfacing TM to RCU readers [54, Section 16.2.3.3], including delaying RCU readers to avoid confilicts, converting RCU readers to transactions, and finally restricting RCU to linked structures and allowing RCU readers to see pre-commit values but waiting for a grace period before freeing data allocated by aborted transactions. There is also the null strategy of prohibiting combining TM with RCU.

Matveev *et al.* [46] introduce an RCU-like mechanism called read-log-update

(RLU) which is an STM with explicitly marked readers. RLU provides good update-side performance and reasonable scalablility, at least for an STM [40]. RLU also provides the simple programmability expected from an STM but with the added requirement that the developer explicitly mark RLU readers. However, RLU incurs read-side overhead that is unacceptable for some workloads [40], a result corroborated by our work.

Siakavaras *et al.* [52] develope a concurrent binary search tree that is protected by a combination of RCU and HTM (RCU-HTM). Because RCU-HTM readers are exactly the same as those of RCU, RCU-HTM enjoys excellent read-side performance and scalability. Updaters synchronize among themselves using HTM. Updaters retry upon transaction failure, but if the number of retries exceeds a predefined limit, the updater falls back to locking. Updaters thus avoid locking if their transactions succeed and in that happy situation provide good performance and scalability. Finally, HTM provides good update-side programmability. However, we will show that RCU-HTM does not address NUMA performance issues.

Figure 3.2 compares the performance of RCU, RLU, and RCU-HTM on linked lists with 256 initial nodes for a varying number of threads for both read-only and 20%-update workloads. The evaluation was peformed on an Intel Xeon system with 4-way 18-core hyperthreading for a total of 144 hardware threads. For a detailed setup of evaluations, see Section 3.4.1. Figure 3.2(a) shows that RLU fails to provide good read performance due to its log-lookup overhead, which discourages the use of RLU for read-mostly workloads. RCU-HTM provides almost ideal read-side performance and scales relatively well, while 20% updates are imposed. However, Figure 3.2(b) shows that RCU-HTM's update-heavy scalability collapses after 36 threads. RLU's update-side scalability degradation is less dramatic but still decreases to that of RCU-HTM at 126 threads. That said, the update-side performance of both RLU and RCU-HTM beats that of RCU in this case due to the single update-side lock.

Modern operating system kernels hold a per-process reader-writer lock in read mode across each page fault in order to ensure that virtual-memory layout does not changed, and this coarse-grained locking can result in poor scalability. Zijlstra *et al.* [66] propose the speculative page fault (SPF), which avoid holding this lock. Instead, SRCU [60] is used to guarantee that the mapping descriptor will remain available until the end of the SPF handler. SRCU also permits readers to block, thus accommodating page faults requiring I/O. The Linux kernel community has continued work on SPF [67]. Clements *et al.* achieved similar effects by replacing the Linux-kernel memory-mapping data structure with an RCU-protecteda binary tree [68] (RCUVM) and further develop a scalable virtual memory system on a research kernel [69] (RadixVM).

3.3 An RCU Extension for NUMA Systems

3.3.1 Root Cause of HTM Performance Degradation on NUMA systems

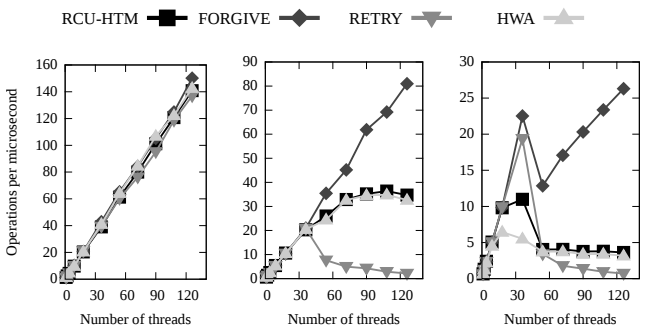
As frequent transaction aborts can severely disturb progress, HTM users should provide an efficient fallback mechanism and minimize the number of aborts. However, RCU-HTM neither provides an optimal fallback mechanism nor minimizes aborts for read-mostly workloads.

If an HTM transaction aborts, RCU-HTM retries the transaction until it success but falls back to traditional locking if the number of retries exceeds a predefined limit. Default value for the limit in RCU-HTM is 10. This fallback mechanism is NUMA-oblivious, and the fixed limit cannot be optimal for dynamic workloads. To compare and determine the effect of various fallback mechanisms, we designed three straightforward fallback mechanisms for RCU-HTM. The first one (FORGIVE) just forgives the operation. If the transaction aborts, it just returns a failure message

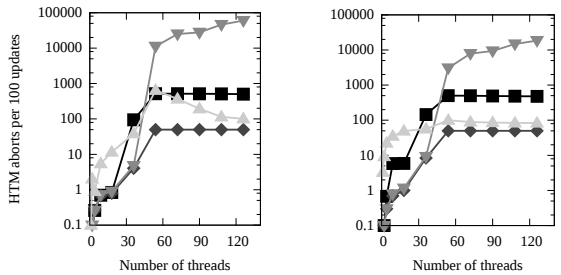
to the user and processes the next operation for different data items. We can expect this mechanism to provide near-optimal performance because it imposes nearly zero fallback overhead. Of course, it cannot be adopted by general applications because the applications should have additional handling of the failure. The second one (RETRY) retries the transaction until success without the limit. This mechanism will have reasonable performance if the data conflict probability is sufficiently low. The third fallback mechanism (HWA) retries the aborted transaction based on the advice from the HTM. For every transaction abort, Intel's HTM provides an advice that indicates whether a retry of the transaction is likely to be successful or not. Because the hardware contains internal information, relying on this hardware advice can be reasonable, although the advice cannot be always right. HWA just follows the advice to decide whether to retry or to fall back to locking.

We implemented three RCU-HTM variants using the three different fallback mechanisms and evaluated their performance with linked lists and hash tables in kernel space. Refer to Section 3.4.1 for a more detailed setup.

Figure 3.3 shows throughput and the number of aborts per 100 updates of the variants for linked lists with 256 initial nodes. Note that the y-axis for abort rates is in log-scale. The variants shows only subtle differences for read-only workload (Figure 3.3(a)) because every variant use the RCU original read. With updates and a large number of threads, FORGIVE performs the best, followed by RCU-HTM and HWA. RETRY has the worst performance due to high abort rates. The performance and abort rates of FORGIVE and RETRY are clearly in inverse proportion. Contrarily, those of RCU-HTM and HWA is not in inverse proportion. RCU-HTM slightly outperforms HWA though it has about 5 times higher abort rates with 126 threads. This comes from the fact that the fallback mechanism of these variants is a global locking, which is expensive and cannot scale. In short, HWA's effect of the reduced aborts is ignored due to the high cost of fallback.



(a) Read-only (b) 2% updates (c) 20% updates



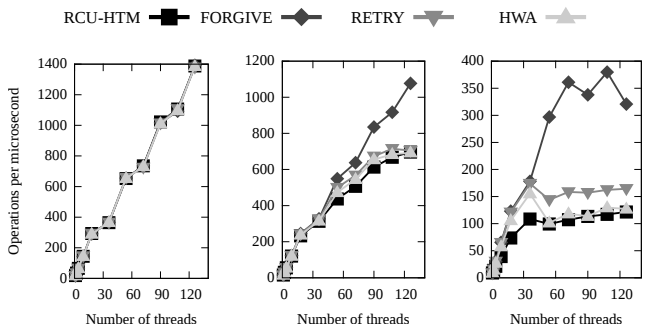
(d) 2% updates (e) 20% updates

Figure 3.3 Performance and HTM abort rates of RCU-HTM variants for linked lists with 256 initial nodes.

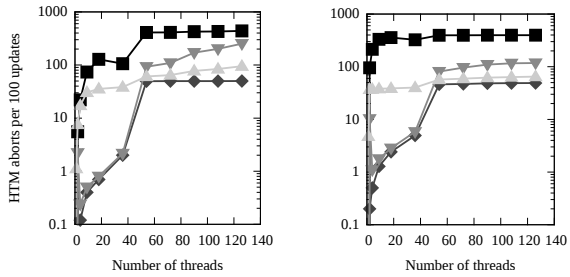
Figure 3.4 shows the performance and number of aborts per 100 updates again for hash tables with 128 buckets and 8 initial nodes per bucket. Compared to the previous evaluation, conflict probability has reduced because the total number of data items has increased from 256 to 1024. In this case, the order of RETRY, HWA, and RCU-HTM has inverted, while FORGIVE still performs the best. The significantly reduced aborts of RETRY compared to the linked list case explains the inversion. It outperforms HWA even with its higher abort rates because it has no expensive locking based fallback. HWA slightly outperforms RCU-HTM though the gap of abort rate between them is almost same, because the cost of the fallback for hash tables, locking per bucket, is lower than that of linked lists, global locking.

Every variant except RETRY fails to scale after 36 threads. Because the evaluation pins the threads to CPU cores so that a minimal number of NUMA nodes is used, this result means that those variants fail to scale as soon as remote NUMA nodes are involved. The abort rate also rapidly increases by involvement of remote NUMA nodes (54 threads). We believe that the abort rates are amplified because of the much higher communication latency between CPUs in different NUMA compared to that of CPUs in same NUMA node. The higher latency increases the conflict checking time of each transaction, and the increased transaction lifetime leads to a higher abort possibility.

Interestingly, the abort rates reduces as more updates induced. This is due to the fact that RCU-HTM is not optimized to minimize abort rates for read-mostly workloads. When data conflict occurs between an HTM transaction and a non-transactional (normal) execution, the HTM transaction is aborted for consistent view of the normal execution because the normal execution cannot be aborted. In the case of RCU-HTM and its variants, because updater threads modify the data shared with the reader in HTM transactions, reader threads can abort updater threads. In read-mostly workloads, more updates can be aborted by the large number of reads.



(a) Read-only (b) 2% updates (c) 20% updates



(d) 2% updates (e) 20% updates

Figure 3.4 Performance and HTM abort rates of RCU-HTM variants for hash tables with 128 buckets and 8 initial nodes per bucket.

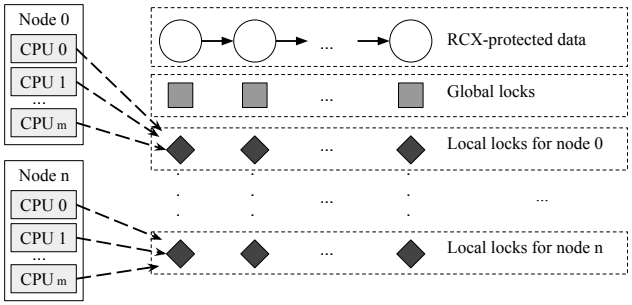
In short, the evaluation results suggest the following conclusions. First, both of high HTM abort rate and expensive fallback damage performance. Second, the HTM abort rate rapidly increases as multiple NUMA nodes become involved. Third, readers can abort HTM transactions of updaters if the data is shared.

3.3.2 Design of RCX

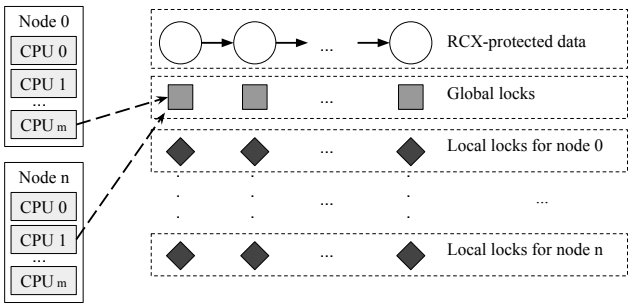
In this section, we provide the design of RCX, which is developed with the conclusions from previous sections. Threads reading data items protected by the RCX use the primitives of the RCU (Listing 3.1). Therefore, RCX imposes no additional overhead on reader threads and provides the almost ideal performance of reads.

Meanwhile, RCX updater threads should follow the updater-side synchronization process of RCX. Initially, every data item protected by RCX should embed a few additional variables. The variables are one global lock and per-NUMA node locks. These variables should be aligned by a cache line size so that access to one of them does not have any effect on the other. In other words, each data item has at least number of nodes plus one additional cache lines. The synchronization process consists of two phases. The first phase elects leader threads of each NUMA node (Figure 3.5(a)). An elected leader is the only one thread on its NUMA node that is trying to update the given data items. In detail, a thread holding the per-NUMA node locks of given data items for the thread's NUMA node is elected as a leader. We use HTM transactions for acquisition of the locks in this phase. The second phase is global locking between the elected threads (Figure 3.5(b)). We use a traditional software-based spinlock for this part. Thus, only one thread in the system can pass the process and exclusively modify the given items.

The process means that each update should acquire two kinds locks. Compared to other mechanisms requiring single-lock acquisition, this could be seen as additional overhead. However, the additional lock acquisition overhead is subtle because the



(a) Node leader election



(b) Global locking

Figure 3.5 Hierarchical synchronization of RCX.

hierarchical synchronization considerably reduces contention for each lock. Figure 3.5 assumes a system utilizing n NUMA nodes and m CPU cores in each node. If the mechanism is based on single-lock acquisition or a single HTM transaction, $n \times m$ threads contend in the worst case. However, in our hierarchical synchronization, worst case contention for the per-NUMA node lock and the global lock is limited to m and n, respectively.

The per-NUMA node locks are contended by threads on the same NUMA node. Therefore, HTM transaction abort amplification that is incurred by an involvement of remote NUMA nodes is avoided. Also, the aforementioned evaluation results for RCU-HTM variants show that HTM scales well with threads in the same NUMA node. That's why we use HTM for the per-NUMA node lock acquisition. The number of nodes (m) is usually much smaller than that of CPU cores in each node (n) for common multicore systems. The contention for the global lock acquisition is thus much smaller than that of per-NUMA node locks. Therefore, the use of a traditional spinlock for the global lock is efficient.

Finally, the per-NUMA node lock, which is the only data that HTM transaction accesses, is isolated from the RCX readers. This isolation avoids the HTM transaction aborts incurred by readers in read-mostly workloads (see Section 3.3.1 for details).

```
1 struct entry {
2     int val;
3     node_t *next;
4     int removed;
5     struct rcu_head rcu;
6     cacheline_t pnodelocks[nr_numa_nodes];
7     spinlock_t global_lock;
8 } entry_t;
```

Listing 3.3 RCX protected data item.

```
1 #define pndlock(e)                          \
2     (e->pnodelocks[numa_node_id()])
3 #define pnd_lock(e) pndlock(e) = 1;
4 #define pnd_ulock(e) pndlock(e) = 0;
5 #define glb_lock(e)                         \
6     spin_lock(e->global_lock);
7 #define glb_ulock(e)                        \
8     spin_unlock(e->global_lock);
```

Listing 3.4 RCX list: add helper macros.

```
 1 int rcx_list_add(list, val)
 2 {
 3 retry:
 4     prev = rcu_deref(list->head);
 5     next = rcu_deref(prev->next);
 6     while (next->val < val) {
 7         prev = next;
 8         next = rcu_deref(prev->next);
 9     }
10     if (val == next->val) return 1;
11     new_entry = rcx_new_entry();
12     new_entry->val = val;
13     new_entry->next = next;
14     while (pndlock(prev) || pndlock(next));
15     if (_xbegin() == _XBEGIN_STARTED) {
16         if (pndlock(prev) || pndlock(next))
17             _xabort();
18         pnd_lock(prev); pnd_lock(next);
19         _xend();
```

```
20      } else { /* Transaction aborted */
21          kfree(new_entry);
22          goto retry;
23      }
24      glb_lock(prev); glb_lock(next);
25      if (prev->removed || next->removed ||
26              rcu_deref(prev->next) != next)
27          goto unlock_retry;
28      rcu_assign(prev->next, new_entry);
29      success = 1;
30  unlock_retry:
31      glb_ulock(next); glb_ulock(prev);
32      pnd_ulock(prev); pnd_ulock(next);
33      if (success) return 0;
34      goto retry;
35  }
```

Listing 3.5 RCX list: add function.

In addition to the logical description, we present an implementation of an RCX-protected sorted linked list with pseudo-code. A pseudo-code for the insertion of an item into the list is shown in Listings 3.3, 3.4, and 3.5. Listing 3.3 shows the data structure for the list, which is embedding the additional variables for RCX (`pnodelocks` and `global_lock`), which are cache-aligned, and Listing 3.4 shows helper macros for redundant code. The core logic for the insertion is in Listing 3.5. Once an RCX updater thread finds data items to update (lines 4–10), it should prepare a new item to insert (lines 11–13) and acquire per-NUMA node locks for the items. The thread first checks whether the locks are free and spins on the locks until the locks are marked as free (line 14). If the locks are all free, the thread starts a hardware transaction (line 15). In the transaction, it checks whether a concurrent transaction

has already acquired the locks before this transaction starts, and if so, aborts this transaction (lines 16–17). If not, it acquires the locks and completes the transaction (lines 18–19). Because both lookup and acquisition of the locks are done in an HTM transaction, only one thread in the NUMA node can hold the lock. If the transaction is aborted, it means that there was a concurrent transaction to the data items and that the locks are not acquired by this thread. The thread should then free the allocated memory and restart by finding items to update (lines 20–23). If the transaction is successful, it means the thread is the only thread holding the per-NUMA node locks for updating items in the current NUMA node. This thread then acquires the global locks of the updating items (lines 24). After this acquisition, it checks whether there were concurrent updates to the target data items (line 25 and 26). If so, it releases every lock it acquired and restarts by finding items to update (line 27 and lines 30–34). If the global locks are acquired and no other concurrent updates to given data items ocurred, the thread can safely update the items using the pointer assignment and necessary memory barriers (`rcu_assign()`) (line 28). Once the update is done, it releases every lock it acquired and returns (line 29–33).

Listings 3.6 and 3.7 show pseudo code for the removal of an item from the list. We skip detailed description of the code because the basic process is the same for the insertion, although it has additional tasks including data item invalidation (line 31).

```
1 #define pnd_locks(e1, e2, e3)          \
2     pnd_lock(e1); pnd_lock(e2);        \
3     pnd_lock(e3);
4 #define pnd_ulocks(e1, e2, e3)         \
5     pnd_ulock(e1); pnd_ulock(e2);      \
6     pnd_ulock(e3);
7 #define glb_locks(e1, e2, e3)          \
8     glb_lock(e1); glb_lock(e2);        \
9     glb_lock(e3)
```

```
10 #define glb_ulocks(e1, e2, e3)        \
11     glb_ulock(e1); glb_ulock(e2);     \
12     glb_ulock(e2);
```

Listing 3.6 RCX list: remove helper macros.

```
 1 void rcx_list_remove(list, val)
 2 {
 3 retry:
 4     p = rcu_deref(list->head);
 5     n = rcu_deref(p->next);
 6     while (n->val < val) {
 7         p = n;
 8         n = rcu_deref(p->next);
 9     }
10     if (val != n->val) return 0;
12     e = n;
13     n = rcu_deref(n->next);
13     while (pndlock(p) ||
14             pndlock(e) || pndlock(n));
15     if (_xbegin() == _XBEGIN_STARTED) {
16         if (pndlock(p) ||
17                 pndlock(e) || pndlock(n))
18             _xabort();
19         pnd_locks(p, e, n);
20         _xend();
21     } else { /* transaction aborted */
22         goto retry;
23     }
24     glb_locks(p, e, n);
25     if (p->removed || e->removed ||
```

```
26              n->removed)
27          goto unlock_retry;
28      if (rcu_deref(p->next) != e ||
29          rcu_deref(e->next) != n)
30          goto unlock_retry;
31      rcu_assign(p->next, n)
32      e->removed = 1;
33      kfree_rcu(e, rcu);
34      success = 1;
35  unlock_retry:
36      glob_ulocks(n, e, p);
37      pnd_ulocks(n, e, p);
38      if (success) return;
39      goto retry;
40  }
```

Listing 3.7 RCX list: remove function.

Note that RCX provides no software handler for situations involving endless HTM aborts. Because of HTM's transaction-size limit, such a handler is required to guarantee forward progress in the general case. The FORGIVE and RETRY versions of RCU-HTM (Section 3.3.1) also lack such a handler, which can be problematic for transactions with unbounded memory footprints. However, because RCX's HTM transactions access only the per-NUMA node locks of the updated data items, their size is bounded above by the number of updated items times the size of a single cache line. Thus, the limited-size RCX updates in this paper should not fail due to cache overflow. If the number of updated items could exceed the limit, we suggest splitting the items into several groups and acquiring the per-NUMA node and global locks separately for each group. This workaround avoids the problem, though it will incur additional overhead and might be hard to apply for unknown transaction-size limit.

3.3.3 Implementation

For proof of concept and a micro and macro evaluation, we implemented RCX-protected sorted linked lists and hash tables and optimized an operating system kernel using RCX to process a part of the virtual memory management task in parallel. All implementations work in kernel space because RCU works best and most reliably in this space whereas userspace RCU implementations are not optimally tuned and userspace memory allocators often have odd bottlenecks.

The implementation of the linked lists and hash tables is based on the RLU kernel space micro-benchmark implementation source code, which is available at Github (`https://github.com/rlu-sync/rlu`). Based on the code, we implemented linked lists and hash tables protected by RCX, RCU-HTM, and its variants. Refer to Section 3.3.2 for a detailed description of the RCX-protected linked list implementation. The current implementation of RCX-protected hash tables consists of about 200 lines of code. Note that the current implementations are made to be compatible only with Intel Haswell or later architectures, although the idea behind RCX and the implementations are not dependent on specific architectures.

Few approaches for RCU based scalable virtual address space [68, 66] of the Linux kernel have proposed, though the approaches cannot scale with concurrent address space modification due to the coarse grained locking for updates. Upon such an approach, we further modified the virtual memory system to process a few common code paths of address space modification in parallel with RCX protection. In detail, we modified a part of the `mprotect()` system call that adjusts only a part of the address space (virtual memory areas) by applying RCX to protect each virtual memory area. As a result, `mprotect()` system calls to different virtual memory areas can proceed in parallel with the modified kernel. We call the modified kernel RCXVM. For the concurrent address space lookup of RCXVM, we applied the SPF patchset [67]

instead of RCUVM [68] because these two approaches have similar idea and SPF is optimized for recent Linux kernels. In total, the use of RCX modified about 1900 lines of code. However, most other changes were made for code encapsulation rather than core logic. We believe the core logic modification is not so complicated.

As of this writing, we are organizing the source code of the implementations. The source code will be available as free/open-source software as soon as the organization is finished.

3.4 Evaluation

This section provides two kind of evaluations for RCX, micro-scope and macro-scope. For micro-scope evaluation, we evaluate performance and HTM abort rates of linked lists and hash tables protected by the RCU, RLU, RCU-HTM, and RCX with varying update rates and number of threads. On the other hand, for macro-scope evaluation, we evaluate performance of two realistic workloads from MOSBENCH suite [70] on three version of the Linux kernel (original, speculative page faults and RCXVM).

3.4.1 Evaluation Setup

We use a huge NUMA system for our evaluation. The system includes 4-way Intel Xeon E7-8870 v3 processors. Each processor utilizes 18 CPU cores, and each core provides two hardware threads (Intel hyperthread is enabled). In total, the system provides 144 hardware threads. The system has 150 GB DRAM for each NUMA node and a total of 600 GB of main memory. The system is running Ubuntu Server 16.04.3 and the v4.19 Linux kernel.

To minimize measurement errors, we repeat every evaluation five times and use averaged value. Standard deviations for repeated results were negligible.

3.4.2 Micro-benchmarks

To show overall performance and to see whether our intended design works as expected in detail, we use simple data structures. For this evaluation, we measure throughput and HTM abort rates of linked lists and hash tables protected by RCU, RLU, RCU-HTM, and RCX in kernel space.

Hash tables are constructed with multiple buckets, and each bucket is a linked list. In other words, the linked lists are hash tables with a single bucket. The linked lists contain integers in sorted order, as Listings 3.5 and 3.7 describe. The linked list protected by RCU uses global locking. RCU-HTM uses the same global locking for the list when it falls back to locking.

For each evaluation, we specify the desired number of threads, the number of buckets (b), the number of initial items in each bucket (n), and the update rate (u). We construct a hash table instance with b buckets, and insert $b \times n$ random integers ranging from zero to $2 \times b \times n$. For example, if 2 buckets and 10 initial items per bucket are desired, we insert 20 integers ranging from zero to 40. After the initialization, we induce lookup, insert, and remove operations with random keys ranging from zero to $2 \times b \times n$ for 3 seconds. After the 3 seconds, we measure the number of successfully processed operations per microsecond and the number of HTM aborts per 100 updates. Insertions and removals are induced with rate $u \div 2$. For example, if u is 20%, 80% of operations do lookup, 10% do insert, and the remaining 10% do remove.

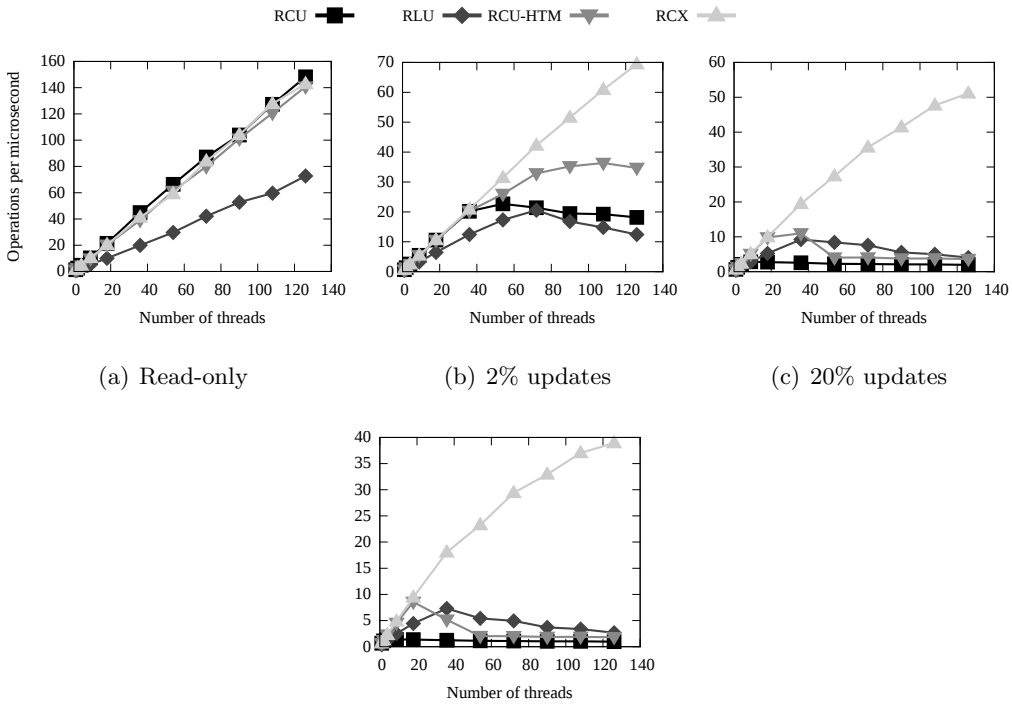
We pin working threads to CPU cores to use a minimal number of NUMA nodes and vary the number of working threads to 1, 2, 4, 9, 18, 36, 54, 72, 90, 108, and 126. For example, if the number of threads is 36, every hardware thread in the first NUMA node works. If the number is 54, every hardware thread in the first NUMA node and 18 hardware threads in the second NUMA node work.

Figure 3.6 shows the performance of the linked lists with 256 initial nodes for read-only, 2%, 20%, and 40% updates. With read-only workload (Figure 3.6(a)), RCU performs best and scales with almost ideal performance. RCX and RCU-HTM also shows ideal performance similar to RCU because their read operations are exactly the same as those of RCU. However, RLU shows a clear overhead due to the log-seeking overhead. With 2% updates (Figure 3.6(b)), RCU stops scaling from 54 threads. RLU scales further until 72 threads, but the performance clearly decreases after that point. RCU-HTM keeps scaling until 108 threads, but the performance decreases slightly with 126 threads. As update rates increase (Figure 3.6(c) and 3.6(d)), every linked list except the RCX-protected one stops scaling and even collapses as the number of threads increases. However, RCX keeps the best performance and scaling towards 126 threads.

Figure 3.7 shows abort rates of the linked lists for the workload above. Because RCU and RLU do not use HTM at all, the variants always show 0% abort rates. In addition, we skip the measurement for read-only workloads because there is no update at all. Note that the y-axis is in log-scale.

As the previous evaluation of RCU-HTM variants also shows, the abort rate of RCU-HTM rapidly increases as soon as a remote NUMA node is involved (after 36 threads). The rate can grow up to 1000% because RCU-HTM retries aborted transactions up to ten times and then falls back to traditional locking. The abort rate for RCU-HTM grows as the number of threads increases but stops growing above 500% from 54 threads regardless of update rates. In other words, as soon as a remote NUMA node is involved, RCU-HTM fails to process the updates in HTM transactions about five times for each update. Meanwhile, RCX maintains only up to 10% abort rate for every case regardless of update rates due to its NUMA-aware, safe-from-read synchronization.

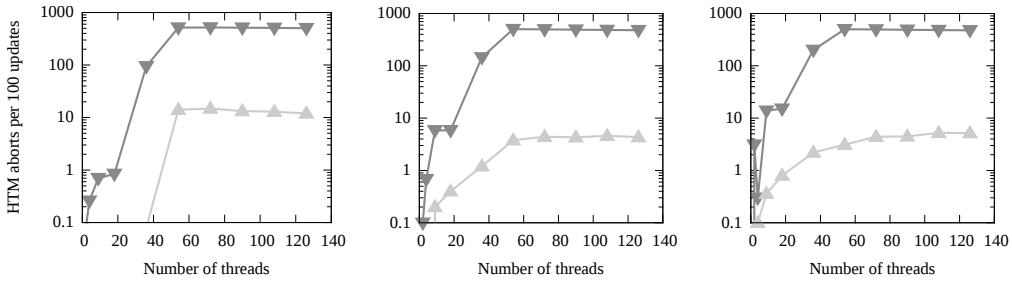
Figure 3.8 shows the performance of hash tables with 128 buckets and 32 initial



(a) Read-only (b) 2% updates (c) 20% updates

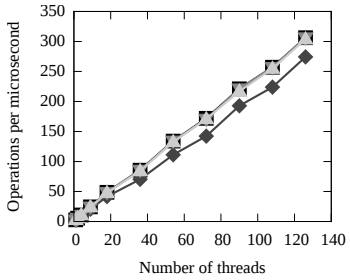
(d) 40% updates

Figure 3.6 Performance of RCU variants for linked lists with 256 initial nodes.

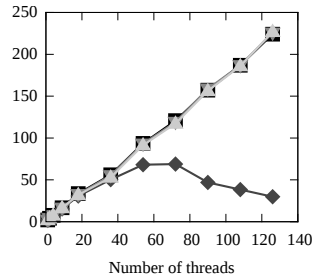


(a) 2% updates (b) 20% updates (c) 40% updates

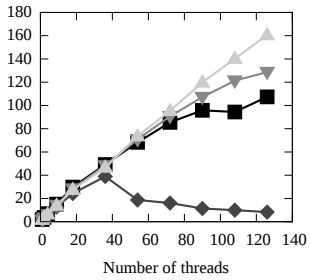
Figure 3.7 HTM abort rates of RCU variants for linked lists with 256 initial nodes.



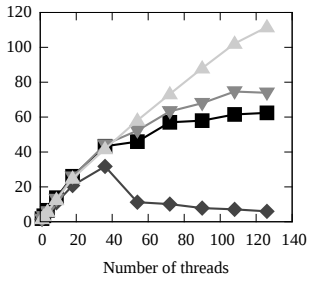
(a) Read-only



(b) 2% updates

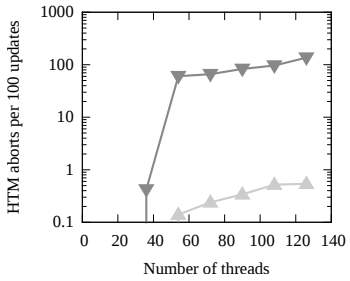


(c) 20% updates

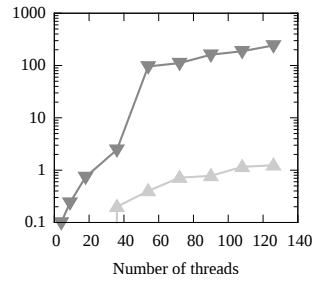


(d) 40% updates

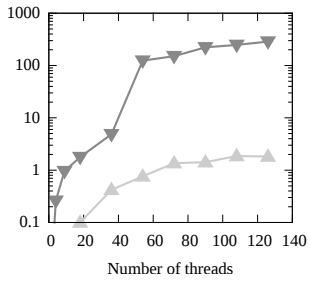
Figure 3.8 Performance of RCU variants for hash tables with 128 buckets and 32 inital nodes per bucket.



(a) 2% updates



(b) 20% updates



(c) 40% updates

Figure 3.9 HTM abort rates of RCU variants for hash tables with 128 buckets and 32 initial nodes.

nodes for varying update rates and numbers of threads. Because the number of items is 64 times bigger than that of previous evaluations of linked lists, the data conflict rate is significantly lower. In particular, because the RCU-protected hash table is doing locking per bucket, it can scale with updates in this workload.

The read-only results are similar to those of linked lists. RLU shows lookup overhead, while RCU, RCU-HTM, and RCX show ideal scalability. Due to the hash table scalability, RCU, RCU-HTM, and RCX show similar and almost ideal performance with a 2% update rate. However, RLU does not scale above 72 threads, even in this case.

Nevertheless, the performance of RCU and RCU-HTM gradually decreases and there is a clear performance gap with RCX as soon as multiple NUMA nodes are involved (Figure 3.8(d)), although both RCU and RCU-HTM keeps scaling toward 126 threads. Contrarily, RCX has the best performance and scalability for every case. With 40% update rates and 126 threads, the RCX-protected hash table outperforms RCU-HTM by 1.5 times.

Figure 3.9 shows abort rates for the hash tables workloads. The results are similar to those in Figure 3.7, but the rapid abort rate increases with the involvement of multiple NUMA nodes is more clear. RCX still shows about two orders of magnitude lower abort rates compared to RCU-HTM.

We can conclude that our synchronization mechanism for RCX on NUMA systems, which is utilizing the hierarchical synchronization and the HTM / locking combination, works as expected. The results for RCX show not only that it has the best performance but also that it keeps scaling with multiple remote NUMA nodes.

3.4.3 Macro-benchmark

The evaluation in Section 3.4.2 shows the usefulness of RCX for widely adoptable simple data structures, linked lists, and hash tables. To further show whether RCX

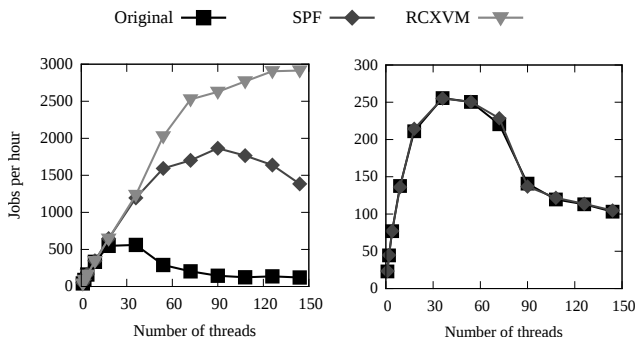
can be used for large, complicated systems and enhance real-world applications, we optimized the v4.19 Linux kernel with RCX for one well-known scalability bottleneck issue [68, 69] in virtual memory address space manipulation for multi-threaded applications. The modified kernel (RCXVM) allows address space adjustment and lookup for typical cases to proceed with concurrent threads in parallel. Nevertheless, note that we parallelized only a few parts of the address space adjustment rather than the entire memory management system. See Section 3.3.3 for implementation details.

MOSBENCH

To show the performance of the optimized kernel for realistic workloads, we run and measure throughput of the *Metis* and the *Psearchy* from the MOSBENCH suite [70], which is developed to compare the scalability of the operating system kernels. Metis is an in-memory multi-threaded map-reduce application that computes a word position index. Psearchy is a parallel version of Searchy [71]. We configure Psearchy to index files in a v4.19.10 Linux kernel source code directory.

Figure 3.10 shows the performance of the applications on three kernels. *Original* is the v4.19 Linux kernel. *SPF* is the same as the Original, but the SPF patchset [67] is applied. *RCXVM* is as described above.

With Original, Metis suffers from bad scalability. SPF significantly improves the performance and scalability of Metis due to concurrent page fault handlings. However, SPF performance also decreases from 90 threads. The scalability bottleneck is the `mprotect()` system call, which updates the protect information of the corresponding virtual memory area (VMA). During the execution of Metis, concurrent user threads intensively invokes `mprotect()`. However, because the Linux kernel protects VMAs of each process with a single reader-writer semaphore (`mmap_sem`), the execution of `mprotect()` becomes a bottleneck when many threads are used. The perf tool [72] based profiling shows an evidence. CPU portion of `mprotect()` kernel space execution



(a) Metis (b) Psearchy

Figure 3.10 Performance of realistic workloads from MOSBENCH on modified virtual memory systems.

is only 2.43% with 36 threads, but grows to 73.22% with 144 threads. Meanwhile, CPU portion for `strcmp()` user space execution is 22.57% with 36 threads, but decreases to only 3.35% with 144 threads. SPF cannot help this problem because the bottleneck is serialized execution of concurrent `mprotect()`, not page fault handling. RCXVM further improves the performance of Metis for many threads due to its address space adjustment parallelization. With 144 threads, RCXVM shows 24.03 times and 2.10 times higher performance compared to Original and SPF, respectively.

Unlike Metis, all three kernels show poor scalability for Psearchy, although SPF and RCXVM show slightly higher performance with 72 threads compared to Original. Our profiling results suggest that the bottleneck of Psearchy is memory mapping incurred TLB flushes. It is widely known that a range of complicated scalability bottlenecks [69] exists in the virtual memory management systems. Because RCXVM is parallelizing only a part of the address space modification and lookup, it cannot address the TLB flush's bottleneck and thus cannot improve the performance of Psearchy.

We do not further apply RCX to optimize the kernel for Psearchy because it is

not within the scope of this paper. Nevertheless, this evaluation result suggests the potential capacity of RCX for operating system kernel scalability optimization.

Web Search Workload (ebizzy)

We additionally evaluate the performance of RCXVM with `ebizzy` benchmark [73], which is designed to resemble common web application server workloads. Each thread receive a request to a certain record and index into the chunk of memory that contains the requested record. Then, the thread allocates a private memory chunk, copy the chunk to the private chunk, and searches the record in the copy using binary search. Again, we repeated each evaluation five times and use averaged value. The deviations are omitted because those were neglibible.

The performance result is shown in Figure 3.11. As similar as the Metis's result, SPF shows highly improved performance while RCXVM further improves it. Original system's performance scales only until 4 threads and then starts to degrade. It shows a performance that is even worse than that of single thread from 36 threads. Meanwhile, SPF keeps scaling until 9 threads with much better scalability. With 9 threads, SPF shows 1.53x performance compared to Original. After that, the performance of SPF starts to rapidly degrade with more threads. This is due to the intensive `mmap()` call of the workload. The workload invoke `mmap()` to allocate the private memory chunk for each record. That said, SPF keeps much higher performance compared to Original with high number of threads. In best case, SPF performs 3.57x better (144 threads). RCXVM keeps scaling until 18 threads but starts dropping the performance due to the intensive `mmap()` invocation with higher number of threads. That said, RCXVM's performance scaling is fastest and its degradation is slowest among the three systems. In best cases, RCXVM performs 2.23x higher performance compared to that of SPF (36 threads), 5.60x higher performance compared to that of Original (72 threads).

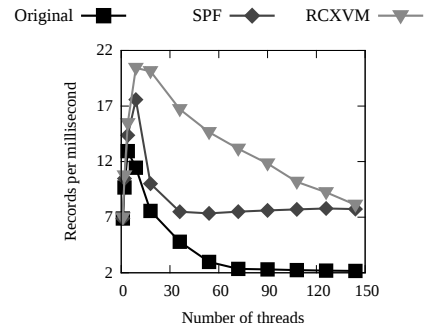


Figure 3.11 Performance of a web server workload on modified virtual memory systems.

3.5 Conclusion

This paper evaluated the performance and properties of state-of-the-art HTM extensions of RCU running on NUMA systems, finding that these extensions all have significant shortcomings. We further investigated the root cause of these shortcomings, and then developed a new synchronization mechanism called RCX that combines RCU and HTM so as to avoid these shortcomings.

We created micro-benchmarks evaluating RCX on simple data structures, resulting in up to 21.54 times higher performance and up to 91.98 times lower HTM transaction abort rates, both compared to a HTM-based state-of-the-art RCU extension. We also carried out a system-level evaluation of RCX by parallelizing a portion of the v4.19 Linux kernel's address-space manipulation, resulting in RCXVM. RCXVM provides up to 40.68 times the performance of the vanilla v4.19 Linux kernel when running a Metis multi-threaded map-reduce application that computes a word-position index. This work clearly demonstrates the performance and scalability benefits of carefully conceived combinations of RCU and HTM, as exemplified by RCX.

Chapter 4

Conculsion

Memory management system is important for performance and scalability of modern workloads because those workloads commonly show memory intensive characteristics. Further, hardware configuration of modern server systems in data center already utilizing huge main memory and large number of CPU cores while memory management systems software is not fully optimized for the trend.

This dissertation finds out and describe in detail about a few parts of modern memory management systems that limits performance and scalability of overall system. Such parts include contiguous memory allocator that provides no guarantee of success and speed of the allocation, the transparent huge pages system that fragile to fragmentation problem, synchronization mechanisms that designed without NUMA systems awareness, and virtual memory system that does not scale with massive threads containing a huge number of memory mappings.

Based on the findings, this dissertation introduced and evaluated two optimization techniques that designed to improve the performance and the scalability of the memory managements systems. For the high performance, a contiguous memory al-

locator guaranteeing success and low latency of allocation has introduced. It achieves the guaranteeness by adopting discardable pages. The adoption of the allocator to the THP systems achieved significant system performance improvement for the memory intensive workloads. Further, this paper introduced a new synchronization mechanism for read-mostly situation, namely RCX, for the high scalability. It achieved the goal by combining the RCU and the HTM in NUMA-aware fashion. Further adoption of this mechanism to the Linux virtual memory system solved a widely-known scalability bottleneck problem and achieved an impressive speedup of an in-memory map-reduce application.

Bibliography

[1] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds," in *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, vol. 47, (New York, New York, USA), p. 37, ACM Press, 2012.

[2] J. Corbet, "Five-level page tables." `https://lwn.net/Articles/717293/`, 2017.

[3] "Intel's new Optane SSDs are superfast and can even work as extra RAM." `https://www.theverge.com/circuitbreaker/2017/10/31/16582018/intel-optane-p900-ssd-fast-dram-nand-flash-memory-desktop-computer`.

[4] "Samsung unveils world's largest SSD with whopping 30TB of storage." `https://www.theverge.com/circuitbreaker/2018/2/20/17031256/worlds-largest-ssd-drive-samsung-30-terabyte-pm1643`.

[5] "Google introduces Android Go, a mobile OS for entry-level devices." `http://bgr.com/2017/05/17/google-io-android-go-release/`, 2017.

[6] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *12th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 16), (Savannah, GA), pp. 705–721, USENIX Association, 2016.

[7] M. Nazarewicz, "Contiguous memory allocator," in *Proceedings of LinuxCon Europe 2012*, LinuxFoundation, 2012.

[8] E. Upton, "Raspberry Pi 2 on sale now at $35." `https://www.raspberrypi.org/raspberry-pi-2-on-sale/`, 2015.

[9] S. Park, "Introduce gcma." `http://lwn.net/Articles/619865/`, 2014.

[10] C. Lameter, "User space contiguous memory allocation for DMA." `https://www.linuxplumbersconf.org/2017/ocw/proposals/4669`, 2017.

[11] "Redis Latency Problems Troubleshooting." `https://redis.io/topics/latency`.

[12] "Disable Transparent Huge Pages (THP) – MongoDB Manual 3.0." `https://docs.mongodb.com/v3.0/tutorial/transparent-huge-pages/`.

[13] V. Babka, "[PATCH v5 6/8] mm, thp: remove __gfp_noretry from khugepaged and madvised allocations." `https://marc.info/?l=linux-kernel&m=146908669009067&w=2`.

[14] J. Araujo, R. Matos, P. Maciel, R. Matias, and I. Beicker, "Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure," in *Proceedings of the Middleware Industry Track Workshop*, pp. 1–7, ACM Press, 2011.

[15] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: a mechanism for speculative address translation," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 307–318, ACM, 2011.

[16] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 258–269, IEEE Computer Society, 2012.

[17] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 237–248, ACM, 2013.

[18] A.-W. Robert Love, "The buffer cache," in *Linux-Kernel Manual: Guidelines for the Design and Implementation of Kernel 2.6*, p. 348, 2005.

[19] J. Corbet, "Cleancache and Frontswap." `http://lwn.net/Articles/386090/`, 2010.

[20] D. Magenheimer, "Transcendent memory in a nutshell." `http://lwn.net/Articles/454795/`, 2011.

[21] M. Freedman, "The Compression Cache: Virtual Memory Compression for Handheld Computers," 2000.

[22] D. Magenheimer, "In kernel memory compression." `http://lwn.net/Articles/545244/`, 2013.

[23] A. Team, "Low RAM," *http://s.android.com/devices/tech/low-ram.html*, 2013.

[24] T. C. Ruth Suehle, "Automatically share memory," in *Raspberry Pi Hacks: Tips & Tools for Making Things with the Inexpensive Linux Computer*, p. 95, 2013.

[25] "Raspberry pi supports cma unofficially." `https://github.com/raspberrypi/linux/issues/503`, 2014.

[26] E. Upton, "Camera board available for sale!." https://www.raspberrypi.org/camera-board-available-for-sale/, 2013.

[27] "Blogbench." http://www.pureftpd.org/project/blogbench.

[28] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis.," *USENIX Annual Technical Conference*, 1996.

[29] "SPEC CPU 2006." https://www.spec.org/cpu2006/.

[30] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation." http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf, 2007.

[31] "TPC-H." http://www.tpc.org/tpch/.

[32] "MariaDB 10.2.8 Release Notes." https://mariadb.com/kb/en/library/mariadb-1028-release-notes/.

[33] T. M. Zeng, "The Android ION memory allocator." http://lwn.net/Articles/480055/, 2012.

[34] J. Stultz, "Integrating the ION memory allocator." http://lwn.net/Articles/565469/, 2013.

[35] N. Willis, "Tizen 2.0 loads up on native code." http://lwn.net/Articles/538888/, 2013.

[36] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng, "DaaC: device-reserved memory as an eviction-based file cache," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems - CASES '12*, (New York, New York, USA), p. 191, ACM Press, oct 2012.

[37] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng, "Rigorous rental memory management for embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 12, p. 1, mar 2013.

[38] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, pp. 509–518, 1998.

[39] D. Bueso, "locking: Introduce range reader/writer lock." `https://lwn.net/Articles/719182/`, 2017.

[40] P. E. McKenney and A. Prasad, "Some more details on Read-Log-Update." `https://lwn.net/Articles/667720/`, 2015.

[41] J. Triplett, *Relativistic Causal Ordering a Memory Model for Scalable Concurrent Data Structures*. PhD thesis, Portland, OR, USA, 2012. AAI3502650.

[42] N. Brown, "The rhashtable documentation I wanted to read." `https://lwn.net/Articles/751374/`, 2018.

[43] M. Arbel and H. Attiya, "Concurrent updates with RCU: search tree as an example," in *Proceedings of the 2014 ACM symposium on Principles of Distributed Computing*, pp. 196–205, ACM, 2014.

[44] N. Brown, "[PATCH 00/20] staging: lustre: convert to rhashtable." `https://lkml.org/lkml/2018/4/11/1341`, 2017.

[45] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pp. 18–32, ACM, 2013.

[46] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: a lightweight synchronization mechanism for concurrent programming," in *Proceedings of the*

25th ACM Symposium on Operating Systems Principles, pp. 168–183, ACM, 2015.

[47] R. Rajwar and M. Dixon, "Intel transactional synchronization extensions," in *Intel Developer Forum San Francisco*, vol. 2012, 2012.

[48] R. Merritt, "IBM plants transactional memory in CPU." `http://www.eetimes.com/electronics-news/4218914/IBM-plants-transactional-memory-in-CPU`, August 2011.

[49] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM System z," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 25–36, IEEE, 2012.

[50] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Washington, DC, USA), pp. 157–168, March 2009.

[51] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *2014 IEEE 30th International Conference on Data Engineering*, pp. 580–591, IEEE, 2014.

[52] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "RCU-HTM: Combining RCU with HTM to implement highly efficient concurrent binary search trees," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–13, IEEE, 2017.

[53] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: expensive synchronization in concurrent algorithms

cannot be eliminated," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 487–498, 2011.

[54] P. E. McKenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It? (First Edition)*. Corvallis, OR, USA: kernel.org, 2014.

[55] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pp. 33–48, ACM, 2013.

[56] S. Kashyap, C. Min, and T. Kim, "Scalable numa-aware blocking synchronization primitives," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017.

[57] H. Han, S. Park, H. Jung, A. Fekete, U. Rohm, and H. Y. Yeom, "Scalable serializable snapshot isolation for multicore systems," in *2014 IEEE 30th International Conference on Data Engineering*, pp. 700–711, IEEE, 2014.

[58] H. Guiroux, R. Lachaize, and V. Quéma, "Multicore locks: The case is not closed yet.," in *Proceedings of the 2016 USENIX Annual Technical Conference*, pp. 649–662, 2016.

[59] W. Deacon, "[PATCH v2 0/5] Get rid of lockless_dereference()." `http://lkml.kernel.org/r/1508840570-22169-3-git-send-email-will.deacon@arm.com`, 2017.

[60] P. E. McKenney, "Sleepable RCU." `https://lwn.net/Articles/202847/`, 2006.

[61] J. R., "Transactional Synchronization in Haswell." `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012.

[62] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: transactional memory for an operating system," *SIGARCH Comput. Archit. News*, vol. 35, pp. 92–103, June 2007.

[63] P. Howard, *Extending Relativistic Programming to Multiple Writers*. PhD thesis, Portland State University, 2012.

[64] P. W. Howard and J. Walpole, "Relativistic red-black trees," *Concurrency and Computation: Practice and Experience*, 2013.

[65] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," *ACM SIGPLAN Noticeis*, vol. 44, pp. 155–165, June 2009.

[66] N. Hussein, "Another attempt at speculative page-fault handling." `https://lwn.net/Articles/730531/`, 2017.

[67] L. Dufour, "[v5,00/22] Speculative page faults." `https://patchwork.ozlabs.org/cover/824425/`, 2017.

[68] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 199–210, 2012.

[69] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 211–224, ACM, 2013.

[70] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, *et al.*, "An analysis of Linux scalability to many cores.," in *OSDI*, vol. 10, pp. 86–93, 2010.

[71] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris, "OverCite: A distributed, cooperative citeseer.," in *NSDI*, vol. 6, pp. 11–11, 2006.

[72] B. Gregg, "perf examples." http://www.brendangregg.com/perf.html.

[73] "Ebizzy." http://ebizzy.sourceforge.net/.