Analyzing The Tradeoff Between Throughput and Latency in Multicore Scalable In-Memory Database Systems

Hitoshi Mitake
Department of Computer Science and Engineering
Waseda University
h.mitake@gmail.com

Hiroshi Yamada
Department of Computer and Information Sciences
Tokyo University of Agriculture and Technology
hiroshiy@cc.tuat.ac.jp

Tatsuo Nakajima
Department of Computer Science and Engineering
Waseda University
tatsuo@dcl.cs.waseda.ac.jp

Abstract

In this paper, we present a tradeoff between throughput and latency in multicore scalable in-memory database systems by showing the results of a performance evaluation and analysis of Masstree, a state-of-the-art multicore scalable data structure that forms the foundation of a variety of multicore scalable database systems. The key technique to make Masstree scalable is an advanced concurrency control technique. Such a technique reduces cache line contention between cores and contributes to high throughput and scalability. However, surprisingly, the worst case latency of the Masstree-based key-value storage system was more than 10 times larger than the score of the memcached system. To detect the main source of the high latency spikes, we analyzed the concurrency control techniques of Masstree. As a result, we found that read-copy update (RCU), an important technique that enables scalability in Masstree, becomes the dominant factor in the high latency spikes. We present a straightforward approach to resolve the latency spikes. In addition, we also show the limitation of the straightforward approach and possible future directions of essential solutions.

1. Introduction

In-memory database systems are key components in today's large-scale internet services. Storage systems that process requests in low latency are becoming critical components for the support of the modern services such as search engines, messengers, and office suites; the deployment of such systems spans from servers in data centers to web browsers and mobile devices. Thus, the sub-milli seconds response is essential for these storage systems [7], and it is important that they reduce disk I/O. Research and development on in-memory caching systems have been ongoing. In addition, recent improvements in DRAM capacity have made it possible for database systems to store all of the data in the main memory [31, 32].

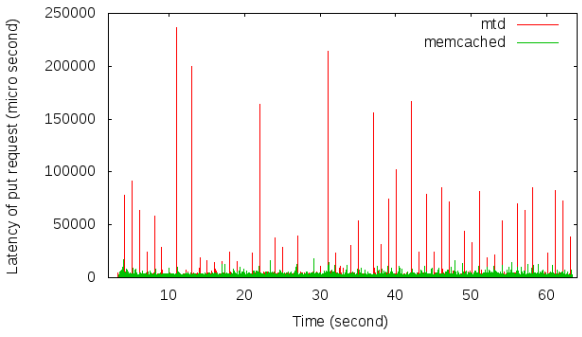

Figure 1. Latency timelines of continuous put requests of two in-memory key value storage systems: memcached and mtd. mtd is built on Masstree. Horizontal axis represents elapsed seconds since starting the benchmark. Vertical axis represents a latency of request processing in micro seconds.

Recently, many studies have proposed in-memory database systems that exploit multiple cores. Early on, in-memory database systems did not use multiple cores because their in-memory design could outperform traditional disk-based systems by 1-2 orders of magnitude without utilizing multiple threads [32]. Although mutex-based coordination between threads was considered as a source of high latency [31], it has become clear that exploiting multiple cores will contributes to improved throughput, load balancing and resource utilization. As a consequence, the recent studies propose the multicore scalable in-memory database systems [19, 20, 24, 26, 33, 35]. For increased the performance, modern multicore processors require less cache line contentions if the cores access shared data structures [5]. There-

APSys '16, August 04-05, 2016, Hong Kong, China

ACM 978-1-4503-4265-0/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2967360.2967361

fore, such systems are built on advanced concurrency control techniques instead of naive locks. These techniques are especially important for realizing scale-up systems, because the latter do not perform well with the addition of new nodes as in the case of scale-out systems [8, 11], but instead must exploit the processor cores of a single machine.

Masstree is a pioneer data structure that adopts advanced concurrency control techniques [26]; it is a B+ tree based data structure that can form the foundation of scalable key-value stores and RDBMSes. The concurrency control techniques of Masstree combines optimistic concurrency control (OCC) [9, 21] and read-copy update (RCU) [28] to allow multiple cores to access a single data structure with fewer cache line contentions. In the case of key-value storage, the design of Masstree enables much higher IOPS than traditional systems [26]. In the case of database systems, many succeeding research systems [16, 19, 33, 35] and a production storage system for data analytics [25] employ Masstree or its variant for multicore scalability. To the best of our knowledge, Masstree is the only one data structure that can support scale-up RDBMSes. Other data structures, such as the hash table used in MICA [24], may not support RDBMSes well because it lacks the support of range queries.

The systems showed that the advanced concurrency control techniques can enable multicore scalability and high throughput. However, to the best of our knowledge, its effect on latency has not been sufficiently analyzed in spite of its importance [7]. For the analysis, we evaluated the throughput and latency of mtd [4], a key-value store system that is built on Masstree, and we used our own modified mutilate [23, 29], originally designed for evaluating the throughput and latency of memcached, that supports the mtd protocol for the benchmarking[1]. Figure 1 shows latency timelines of memcached and mtd under continuous put requests (the detailed configuration and parameters of the benchmark are described in Section 3). As the figure depicts, mtd results in a number of extremely high latency spikes. We believe that it is important to analyze and understand the behavior of Masstree because the end-to-end low latency becomes impossible despite systems level techniques for reducing latency [1, 3, 14] being used when the data structure contains sources of the latency spikes.

We analyzed the latency of Masstree in depth because we thought the result showed important performance properties of the data structure. The most important aspect of Masstree that cannot be found in traditional database systems is its concurrency control techniques. Therefore, we formed our first hypothesis: at least one of the techniques is introducing the high latencies. As described later, we found that the dominant latency factor comes from RCU, specifically the pause time caused by the garbage collection. We believe this result is very useful for improving scale-up in-memory database systems because Masstree forms the foundation of these systems and RCU is a key technique with no promising alternative.

The contributions of the paper are as follows:

- Detailed analysis reveals a drawback of RCU for in-memory database systems; i.e., high latency spikes are shown.
- A simple solution is designed and implemented that can resolve the high latenciey. In addition, we analyze the solution and introduce insights that call attention to its limitations.

The rest of this paper is organized as follows: Section 2 introduces related work. Section 3 shows the analysis of Masstree for detecting the source of high latencies. Section 4 describes why RCU produces high latency spikes. Section 5 presents a straightforward approach to overcome the problem and its evaluation result. We also claim its potential pitfalls and possible future direction to overcome the pitfalls. Section 6 concludes this paper.

2. Related work

2.1 In-memory database systems

H-Store is an RDBMS that stores all data in the main memory [32]. As a result of this design, H-Store achieves a much higher transaction throughput than traditional disk-based RDBMSes. However, it does not exploit to use multiple cores: It uses a partitioning approach for multithreading. Mapping between the keys and cores requires a static configuration. This approach provides neither efficient resource usage nor good load balancing.

Silo is a scale-up transactional RDBMS that overcomes the limitation of the partitioning approach based on Masstree [33]. To exploit multiple cores without partitioning, it was necessary to let the cores share single data source with less cache line contention for scalability. Silo achieves this goal by using Masstree as its index structure and its scalable transaction commit protocol. Actually, Silo provides much higher transaction throughput than H-Store. The design of Silo had a marked influence on succeeding research systems [19, 20, 35]. Silo and these systems are based on RCU-based concurrency control because they use Masstree or its variant data structures. Therefore, they may experience latency spikes as introduced in this paper. Although the initial goal of these systems is high throughput, we believe that high latency spikes will be problematic.

2.2 Scale-out distributed database systems

Scale-out distributed database systems have achieved substantial success [8, 11]. In this area, RAMCloud is a representative system that achieves low-latency request processing capability [18, 31]. The main benefit of scale-out systems is their flexibility: Storage areas, IOPS and bandwidth can be easily enhanced by adding new nodes to the systems. How-

[1] Our modified mutilate used for the evaluation is publicly available at https://github.com/mitake/mutilate

ever, supporting general purpose transaction processing in this kind of systems is known to be difficult and has emerged as a topic of great interest to researchers [22] because both of hardware faults and planned maintenance can result in node join/leave events. In such an environment, maintaining the consistency of distributed objects is a difficult task. There are no proven methodologies that support general purpose distributed transactions with a few exceptions [6].

Our primary target is a technique for scale-up storage systems. Unlike the scale-out systems, the scale-up systems perform by exploiting a single high performance machine with multiple cores to access a shared memory. Therefore it is difficult to scale up the systems without increasing cache contentions. However, supporting rich data types and general purpose transaction processing in the scale-up systems is known to be easier than in the scale-out systems. Actually, Silo and its succeeding systems are considered as promising candidates for this area of research.

3. Analyzing the sources of the high latency spikes

This section analyzes the sources of the high latency spikes that appear in Figure 1. As noted earlier, our hypothesis is that at least one of the concurrency control techniques, OCC and RCU, is the dominant factor in the latency spikes because of the following reasons:

- OCC can introduce reader-side starvation because of the retry by concurrent modification of the writer [19, 21, 35].
- RCU can introduce pause time in garbage collection (GC) [10, 15, 28].

In the experiment, we used Google Compute Engine: 1 VM (32 vCPUs, 28.8GB memory) for the servers and 16 VMs (4 vCPUs, 8GB memory) for mutilate. All VMs run Linux 3.16.0. The VMs for mutilate send requests to the server for 60 seconds intervals. The number of keys is 10000. The length of the keys is 30 bytes, and the length of the values is 200 bytes. These are the default configurations of mutilate. Note that our purpose is to understand the performance properties of the data structure. Therefore, the load is tuned to stress the essential part and avoid the bottlenecks of OSes and hypervisors [3, 17, 29], so the IOPS scores of mtd are lower than the scores reported by Mao et al [26]. In addition, the persistence feature of mtd is disabled to avoid disk I/O overhead. Actually, as we show in the below, the benchmarking result of memcached on the same environment did not show any unreasonable latency spikes so the overhead that comes from the components including the virtualization layer can be ignored.

Table 1 shows the result of the entire experiments. The throughput score of mtd is far better than that of memcached. However, the latency of mtd shows several very high spikes. In addition, the worst case latency of mtd is larger than 230ms (more than 10x larger than that of memcached). This is an unacceptably high latency for in-memory key-value storages, and it has a significant impact in real-world use cases [12].

3.1 Excluding the possibility of latency spikes other than concurrency control

First, we benchmarked both memcached and mtd with read-only traffic to validate our hypothesis. The read-only traffic does not generate the latency spikes due to the concurrency control techniques used in Masstree, so it must not produce latency spikes as a result of these techniques for the following reasons: 1) From the perspective of OCC, the read-only traffic does not modify the data structure, so there is no reader-side retry or starvation, and 2) from the perspective of RCU, the read-only traffic does not produce obsolete objects by updating operations, so garbage collection is not needed. In addition, if the dominant factors are the result of the implementation of mtd (event loop construction, thread scheduling, etc.), the score of mtd should show the high latency spikes even in this benchmark.

The 1st and 2nd rows of Table 1 show the results (note that they do not include the latencies of put requests for initialization). There is no meaningful difference; therefore, the factors listed above are not producing the high latency spikes. This supports our hypothesis that the sources of the latency spikes originate from the concurrency control techniques in Masstree.

3.2 Do the latency spikes originate from OCC?

We showed that components other than concurrency control do not produce the latency spikes. Next, we investigate the OCC usage in Masstree.

Masstree employs OCC to reduce cache line contention produced by reader threads. A conservative approach for hiding an inconsistent state of the data structure during modification by a writer thread from the reader threads is reader writer lock: If the reader threads are reading the data structure, the writer threads are not allowed to modify it. However, the reader threads need to indicate their existence by updating the locks that are shared with the other threads. This operation incurs a great deal of cache invalidation traffic and degrades scalability [5]. Therefore, instead of using the reader writer lock, Masstree employs an OCC-based approach. If the reader threads find that the writer threads are modifying the data in their critical sections, they simply retry their operations from the beginning. This reduces the modification of shared cache lines by reader threads and contributes to scalability.

The above scheme results in the starvation of reader threads [19, 21, 35]. In addition, Masstree uses the user space spinlock for writer-writer coordination. Its design assumes that the typical writer-side operation can be completed quickly. If this assumption is correct, the scheme works very efficiently. However, the preemption of the user

	avg	std dev	min	max	5th	95th	99th	99.9th	99.99th	query/sec
memcached (get only)	422.6	361.9	106.7	17226.0	177.7	968.4	2331.7	3192.1	4654.0	162538.7
mtd (get only)	443.4	370.9	106.7	10422.0	177.3	1025.3	2338.5	3151.5	4756.7	164192.4
memcached (get)	392.4	338.5	106.7	19017.9	177.1	869.5	2272.1	2962.0	4755.0	168242.3
memcached (put)	399.7	336.9	117.4	12991.9	181.7	874.0	2273.2	2989.8	4693.0	
mtd (get, no GC)	371.3	332.0	106.7	10929.1	166.8	815.4	2286.8	2980.4	4758.3	173842.6
mtd (put, no GC)	374.5	335.8	106.7	20639.9	168.7	826.6	2293.9	2977.5	4790.6	
memcached (put only)	483.4	479.3	117.4	18044.0	189.5	1209.0	2517.3	4931.3	10738.1	148011.5
mtd (put only)	418.2	861.3	117.4	236277.8	174.0	941.2	2313.2	3259.1	17862.3	165493.2

Table 1. Comparison of performance of memcached and mtd under various workloads. A unit of the scores that is used other than query/sec is the micro second. nth columns show the average of latencies in the top n percent.

space critical sections is always possible and difficult to control [2]. We thought that the preemption of writer threads can produce high latency spikes in both of get and put requests.

To investigate the effect of writer thread preemption, we benchmarked another benchmark configuration. In this configuration, garbage collection in RCU (another suspected source of the latency spikes, as we describe later) is disabled to focuse on the effects of OCC. 50% of mutilate requests are get requests, and the rest of them are put requests. However, the result did not imply that OCC contributes to the high latency spikes. The 3rd to 6th rows of Table 1 shows the result of this benchmark. No meaningful differences can be found in either of the put or get requests. Therefore, we concluded that the OCC is not the dominant factor in the latency spikes.

3.3 Do the latency spikes originate from RCU?

The last remaining suspect is RCU [28]. RCU was originally designed as an alternative to reader writer lock and atomic reference counting in OS kernels. As we describe later, it can contribute to significant reductions in cache line contention. Masstree uses its variant of RCU for lifetime management of nodes in its tree structure. The nodes of Masstree must be reclaimed when they become obsolete (e.g., deletion or updating). If Masstree uses atomic reference counting for the lifetime management, each lookup operation must incur atomic operations that involve a shared cache line update. Masstree successfully reduces the cache line update with RCU. However, it has an important drawback: garbage collection. Under the lifetime management of RCU, obsolete objects cannot be reclaimed immediately. They are stored in a temporary place for such objects and reclaimed in a batched manner as we describe later. We came to the conclusion that this batched reclamation produced the high latency spikes.

To justify this prediction, we compared the performance of memcached and mtd under the put-only workload. As we described in Section 1, the workload updates a limited number of keys iteratively. Therefore, it produces a large amount of garbage objects and stresses the functionality of GC. The result is shown in the 7th and 8th rows of Table 1. As the score for mtd shown in the 8th row explains, its standard deviation and the worst case latency are quite higher than memcached and mtd under other configurations (the result is a summarization of Figure 1). Based on this analysis, we concluded that RCU is the dominant factor in the high latency spikes.

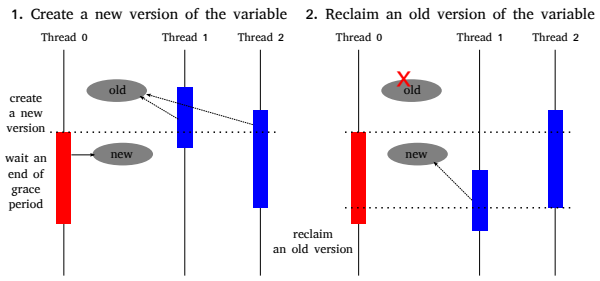

Figure 2. Creation and reclamation of RCU in the kernel space. Red squares indicate writer-side critical sections. A writer side thread cannot finish its critical section until all reader threads finish their critical sections. Blue squares indicate reader-side critical sections. Arrows with the solid line indicate creation. Arrows with the dash line indicate reference. Red x-mark indicates reclamation.

4. Why does RCU produce the latency spikes?

4.1 RCU in an OS kernel space

To understand the reason of why RCU produces such latency spikes, it is necessary to understand the basic mechanism of RCU. As we described in the above, RCU was originally designed as an alternative to reader writer lock and reference counting in the OS kernel space [28]. OS kernels have many data structures that are rarely updated but read frequently in hot paths (e.g., a list of loadable modules). If the hot paths produce invalidation traffic during reader-side locking, the scalability of the kernel is degraded [5].

To avoid producing the traffic of cache coherence protocol in reader-side operations, RCU just requires the reader threads to disable and enable preemption to start and finish their critical sections (in a case of non preemptible kernel, the reader threads need to do nothing). Instead of these ex-

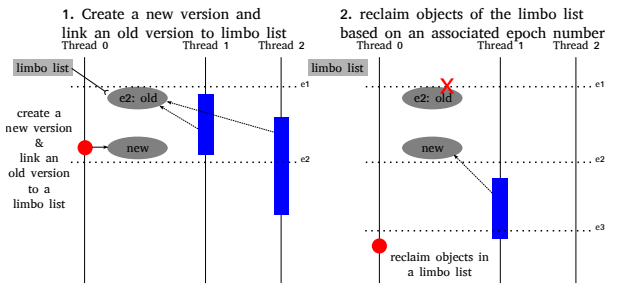

Figure 3. Creation and reclamation of RCU in Masstree (MRCU). A writer thread does not need to wait for the completion of reader side ciritical sections. Reclamations are performed in a batched manner.

tremely simplified and low-cost reader-side operations, RCU introduces complex writer-side operations and limitations to supported data structures [27].

The writer-side operations must consists of the instructions whose effect is visible by other threads in an atomic manner even without locking (e.g., single pointer updates). With this restriction, reader threads cannot observe inconsistent states even without the use of reader writer locking (multiple writers should be serialized with a lock that is acquired only by writers).

The above writer operations can produce multiple versions of the data structure. Even after the writer thread completes the update of the data structure, some reader threads can still be working on the older versions because readers are not serialized with the writer by locking mechanisms. Therefore, the writer thread must wait every reader-side critical sections to avoid memory leaks (this duration is called the *grace period*). To solve this GC problem, RCU exploits the privilege of kernel threads, e.g., activating kernel threads on remote cores with inter processor interrupt. With this privilege, writer threads can detect the end of the all reader side critical sections without explicit communication that involves a shared memory update. It is because the writer threads can have useful invariants about the progress of reader side ciritcal sections e.g., if a kernel thread is scheduled on processor 1, all reader threads that can read an old version variable finished their reader side critical sections on processor 1. Figure 2 presents how RCU works in the OS kernel space.

4.2 RCU in Masstree

Although the original RCU is designed for the OS kernel space, its scalability is also valuable for user space programs. Userspace RCU (URCU) is a project that ports RCU to the user space [34]. URCU implements a mechanism for detecting the grace period with various methodologies including a specialized system call of the Linux kernel. With URCU, user space programs can manage the scalable reader writer coordination even without the privilege that includes the preemption control.

However, this kind of RCU is not enough to achieve concurrency control of a data structure that needs to support high throughput write operations as in Masstree because RCU sacrifices the writer-side performance significantly.

A variant of RCU used in Masstree (we refer to it as MRCU) does not sacrifice the writer-side performance with using a technique called the *epoch based reclamation* (EBR) [10, 15]. Under EBR, a writer thread does not need to wait for the end of a grace period after their modification of the data structure. Instead of waiting for the end of a grace period, the writer thread needs to register an obsolete object (although it can still be read by readers) to a *limbo list* with *epoch number*. The limbo list stores garbage objects that need to be reclaimed after the completion of reading operations. With this mechanism, the writer threads in MRCU can defer reclamation and complete operations without waiting for the grace period as depicted in Figure 3. The garbage objects linked to the limbo list are reclaimed based on their epoch number. The epoch number is a number that is monotonically and periodically increased and visible from every thread. In both of read and write operations, each thread copies the number to its local area that can be visible from every other thread. After the operations, threads read the epoch number of every thread and determine the minimum epoch number that the working threads belong to. After this process, garbage objects whose epoch number is less than the minimum working epoch number can be reclaimed safely (this process is also called *RCU quiesce*). Listings 1 shows a simplified event loop of MRCU executed by every request processing thread (typically one core has one thread).

The reclamation process of MRCU requires a period of time that is proportional to the number of garbage objects that can be reclaimed. The time is almost proportional to the epoch interval. As depicted in Figure 4, shorter intervals result in low throughput and stable latency. In addition, longer intervals result in high throughput and unstable latency. Therefore, the throughput and latency have a relation with tradeoff, and it is affected by an epoch interval. We can obtain lower and more stable latency with a shorter epoch interval, but the throughput may be sacrificed.

Listing 1. Simplified request processing loop of MRCU.

```
1  while (true) {
2    req = receive_request();
3    rcu_start();
4    process_request(req);
5    rcu_end(); // can take hundreds of ms!
6  }
```

4.3 Discussion

Our workload has been simplified for the sake of analysis and does not reflect realistic use cases. Therefore, it needs

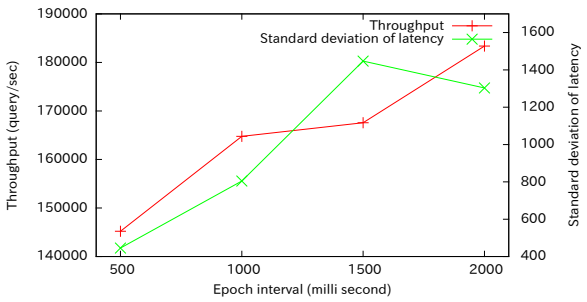

Figure 4. mtd's throughtput and standard deviation of latency under various epoch interval configurations (1000ms is the default configuration)

to be justified to investigate how the high latency spikes can be found and be problematic in realistic use cases. We believe that the latency spikes can be seen in Masstree-based systems under a realistic workloads. For example, a skewed write workload is regarded as a common and challenging workload pattern. The pattern can be found in large-scale web services, e.g., burst requests from like buttons for trending items on SNS and auction services with approaching bidding deadlines. Recent studies have proposed methodologies for this type of workload [24, 30].

Given such a workload, Masstree-based systems will produce a large amount of garbage objects. To clean the objects, long quiesce processes are required, and they result in high latency spikes. Therefore, if the systems are used as components in complex deployments, expensive latency tolerant techniques will be required [7]. In addition, the large amount of garbage objects makes capacity planning of storage space difficult because accurate prediction of the available space for the database systems is required for not exhausting main memory.

Thus, the problem of latency spikes must be overcome to make scale-up in-memory database systems more practical for diverse workloadss.

5. Overcoming the latency spikes problem

This section presents a simple solution to solve the high latency spikes that originate from MRCU. This solution also does not degrade the throughputs of the system, but the solution still has potential pitfalls. After showing the analysis of the potential pitfalls, we present future directions to overcome the pitfalls and make scale-up in-memory database systems more practical for diverse workloads.

5.1 Straightforward solution

The most straightforward solution for reducing the latency spikes is performing the RCU quiesce asynchronously. We implemented this solution and evaluated the effectiveness. In the version that incorporates the solution, mtd creates a thread for RCU quiesce for each request processing thread. Almost every cost of RCU quiesce (produced in Line 5 of Listing 1) can be excluded from a main event loop. We have implemented this solution in mtd and compared the performance with that of memcached and the original mtd. As shown in Table 2, this scheme succesfully reduces the latency spikes with slight throughput degradation.

	avg	std dev	max	99th	query/sec
memcached	483.4	479.3	18044.0	2517.3	148011.5
mtd (orig)	418.2	861.3	236277.8	2313.2	165493.2
mtd (async)	387.3	364.3	18739.2	2264.9	161645.4

Table 2. Comparison of memcached, original mtd and mtd that supports asynchronous RCU quiesce. A unit of the scores that is used other than query/sec is the micro second.

5.2 Potential pitfalls of the simple solution

Although the above solution appears to be effective, we found some potential pitfalls. This needs to introduce new mutexes between the quiesce threads and the request processing thread. Then, at least one mutex is required for a queue that is used to transfer limbo lists between the threads. In addition, if the request processing threads use a per thread object cache for rapid memory allocation (actually mtd uses a per thread slab allocator), it needs to be protected with mutex because both types of the threads can gain access in parallel. This is a classic pitfall of lock-free programming. Even if algorithms do not explicitly require locks, locks in the components used by the algorithm, e.g., a memory allocator, can degrade scalability [10].

The straightforward solution also introduces difficulties of capacity planning. The defered reclamation of garbage objects limits available free space at a given point in time. It is because even if the reclamation is triggered, the reclamation can be delayed for an indefinite period of time. This property makes the capacity planning difficult and require large amount of free space for ensuring that the database systems do not exhaust entire main memory.

In addition, the recent trends in hardware technology imply that this methodology is no longer efficient. Generally speaking, concurrent GC is based on an assumption that most application programs are I/O bound: GC can be executed during network and storage I/O. However, in-memory database systems may not be I/O bound programs essentially.

From the perspective of storage I/O, the request processing in in-memory database systems may become CPU bound. Data is stored in the memory, so all operations consume CPU cycles. In addition, the durable storage used for persistence will also be realized through byte addressable NVRAM [20] in the near future. This means that the persistence process will also consume CPU cycles. From the perspective of network I/O, it is quite common that a traditional interrupt-based approach is not enough to exploit the capability of recent NICs. Virtually every in-memory storage

system that assumes high performance NICs employ polling based network I/O [24, 31].

Therefore, we should not assume that enough CPU cycles can be allocated for asynchronous RCU quiesce of the near future deployments of in-memory database systems.

5.3 Possible approaches to overcome the potential pitfalls

RCU forms the foundation of scale-up in-memory database systems, and there are no other promising alternative concurrency control techniques. Is it impossible to avoid this drawback? We believe that an efficient concurrency control technique for scalable and low-latency in-memory database systems can be constructed. We list possible future directions that offer promises.

Exploiting the properties of balanced trees. It is a basic principle that the access frequency of nodes in the tree depends on their heights. Based on this assumption, we will be able to design a new concurrency control technique that only avoids cache line contention in the access to the nodes near to the root. Cache line contention caused by operation on the nodes near to the leaves will not be expensive possibly because of its low access frequently.

Using scalable reference counting. If we employ a scalable reference counting that enables immediate reclamation [13] for managing a part of the nodes, the size of the limbo list can be kept smaller, and the reclamation time will be shorter.

These hybrid approaches will introduce difficulties as a result of multiple GC algorithms, but we believe the idea is worth to be investigated.

6. Conclusion

In this paper, we analyzed the tradeoff between the performance and latency in Masstree, a data structure that forms the foundation of scalable in-memory database systems. The results in the analysis showed that Masstree-based systems can have high latency spikes potentially. Our analysis of the result revealed that the latency spikes originate from RCU, a key technique for multicore scalability. Although RCU enables scalable lifetime management of database indexes and records, understanding and controlling its drawback are difficult. In the next step, we need to evaluate more diverse workloads and their effects to the drawback of RCU. Based on the experiments, we showed that an efficient technique for reducing a number of garbage objects is required for low latency and scalable in-memory database systems. We hope that our result will design truly robust scale-up in-memory database systems possible. The modified Masstree used in this paper is publicly available at https://github.com/mitake/masstree-beta/tree/latency-analysis

Acknowledgments

We thank to the anonymous reviewers and our shepherd Guangyu Sun for their valuable comments and feedback for improving our paper.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851192. URL http://doi.acm.org/10.1145/1851182.1851192.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi: 10.1145/121132.121151. URL http://doi.acm.org/10.1145/121132.121151.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685053.

[4] Beta release of Masstree. https://github.com/kohler/masstree-beta/. Retrieved May 15, 2016.

[5] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, Jan. 2015. ISSN 0734-2071. doi: 10.1145/2699681. URL http://doi.acm.org/10.1145/2699681.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013. ISSN 0734-2071. doi: 10.1145/2491245. URL http://doi.acm.org/10.1145/2491245.

[7] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794. URL http://doi.acm.org/10.1145/2408776.2408794.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.

1294281. URL http://doi.acm.org/10.1145/1294261.1294281.

[9] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 343–356, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555269. URL http://doi.acm.org/10.1145/2555243.2555269.

[10] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University, 2004. Technical Report UCAM-CL-TR-579.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

[12] GigaSpaces. Amazon found every 100ms of latency cost them 1% in sales. http://goo.gl/BUJgV. Retrieved May 15, 2016.

[13] B. Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 313–321, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X. doi: 10.1145/73141.74846. URL http://doi.acm.org/10.1145/73141.74846.

[14] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 1–14, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. URL http://dl.acm.org/citation.cfm?id=2789770.2789771.

[15] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, Dec. 2007. ISSN 0743-7315. doi: 10.1016/j.jpdc.2007.04.010. URL http://dx.doi.org/10.1016/j.jpdc.2007.04.010.

[16] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 31:1–31:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901348. URL http://doi.acm.org/10.1145/2901318.2901348.

[17] S. Kashyap, C. Min, and T. Kim. Scalability in the clouds!: A myth or reality? In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3554-6. doi: 10.1145/2797022.2797037. URL http://doi.acm.org/10.1145/2797022.2797037.

[18] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kejriwal.

[19] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, 2016. ACM.

[20] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2746480. URL http://doi.acm.org/10.1145/2723372.2746480.

[21] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915. doi: 10.1145/319566.319567. URL http://doi.acm.org/10.1145/319566.319567.

[22] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815416. URL http://doi.acm.org/10.1145/2815400.2815416.

[23] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592821. URL http://doi.acm.org/10.1145/2592798.2592821.

[24] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL http://dl.acm.org/citation.cfm?id=2616448.2616488.

[25] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, and A. Wang. Kudu: Storage for fast analytics on fast data. Technical report, Cloudera, Inc., 2015.

[26] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168855. URL http://doi.acm.org/10.1145/2168836.2168855.

[27] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9.

doi: 10.1145/2815400.2815406. URL http://doi.acm.org/10.1145/2815400.2815406.

[28] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proceedings of Ottawa Linux Symposium*, 2002.

[29] mutilate. https://github.com/leverich/mutilate. Retrieved May 15, 2016.

[30] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 511–524, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685088.

[31] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015. ISSN 0734-2071. doi: 10.1145/2806887. URL http://doi.acm.org/10.1145/2806887.

[32] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325981.

[33] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522713. URL http://doi.acm.org/10.1145/2517349.2522713.

[34] Userspace RCU Project. http://liburcu.org/. Retrieved May 15, 2016.

[35] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang. BCC: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proc. VLDB Endow.*, 9(6):504–515, Jan. 2016. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2904121.2904126.