A High-Throughput In-Memory Index, Durable on Flash-based SSD

Insights into the Winning Solution of the SIGMOD Programming Contest 2011

Thomas Kissinger, Benjamin Schlegel, Matthias Boehm*, Dirk Habich, Wolfgang Lehner
Database Technology Group
Dresden University of Technology
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Growing memory capacities and the increasing number of cores on modern hardware enforces the design of new in-memory indexing structures that reduce the number of memory transfers and minimizes the need for locking to allow massive parallel access. However, most applications depend on hard durability constraints requiring a persistent medium like SSDs, which shorten the latency and throughput gap between main memory and hard disks. In this paper, we present our winning solution of the SIGMOD Programming Contest 2011. It consists of an in-memory indexing structure that provides a balanced read/write performance as well as non-blocking reads and single-lock writes. Complementary to this index, we describe an SSD-optimized logging approach to fit hard durability requirements at a high throughput rate.

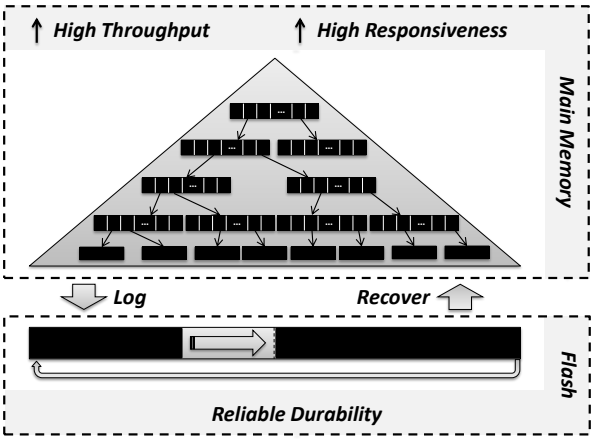


Figure 1: Indexing System Overview.

1. INTRODUCTION

With large main memory capacities becoming affordable over the past years, we observe a shift in the memory hierarchy that degrades hard disks to a persistency-only medium and moves the entire data pool and processing into the main memory. As a second hardware trend, CPU clock rates stopped growing and the number of cores and hardware threads per CPU started to increase constantly. Another present topic are flash-based SSDs, which allow increased throughput and lower latency compared to classic hard disks. When having a look at the application trends, we identify high update rates as a major issue, e.g., in monitoring applications, operational BI or even in social networks. However, common index structures like B+-Trees [2] are designed to work on block-based storage and are not well suited for frequent updates with massive parallelism, because they require complex locking schemes for split and merge operations. This increases communication costs between threads, especially for cache coherency and locking. Furthermore, they are optimized for large block sizes that are used on hard disks. There exist enhancements of the B-Tree that reduce either locking overhead (B-Link Trees [8]) or make them cache-aware (CSB+-Trees [10]). However, these improved structures do not overcome the weaknesses of the B+-Tree base structure in terms of comprehensive balancing tasks and main memory accesses, especially for updates.

In this paper, we describe our solution for the SIGMOD Programming Contest 2011 [1], which addresses exactly the described issues: a high-throughput in-memory index structure, which uses

*The author is currently visiting IBM Almaden Research Center, San Jose, CA, USA.

a flash-based SSD for durability purposes. The task required us to build an in-memory index that fits entirely into the available main memory (two times the total database size) and is able to handle 1024 Byte keys and 4096 Byte values without duplicates. Further, the index needs to offer an order-preserving key-value interface comprising the following operations: (1) *read* the value for a given key, (2) *update* respectively insert a value for a key, (3) *delete* a key-value pair, (4) *compare-and-swap* a value and (5) *iterate* over a key range. The given workload demands a balanced read/write performance as well as a fine-grained locking scheme to allow massive parallel manipulations and reads. For durability, the contest was defining an Intel X25-E enterprise class SSD formatted with ext4. The programming contest constraints granted three times the space of the total database size, which required our solution to perform a continuous garbage collection.

The specifications of the programming contest were released at the end of January 2011 and all teams had about two months available for implementing their solutions. In order to compare the different solutions during this time, the organizers provided a leaderboard, to show the teams each others current results. After the submission deadline was passed, each solution was tested with different workloads (unknown before) to determine the winning team that was finally announced during the SIGMOD 2011.

To give an overview of our winning solution, we illustrate the architecture in Figure 1. The first part of the system forms the index structure that resides completely in the main memory to offer high throughput and low latency to the consuming applications. For our solution, we decided to deploy an enhanced generalized prefix tree [3]. The prefix tree is optimized to work as in-memory structure, because it guarantees a maximum number of memory access for finding a key. In consideration of the workload, this structure also offers a well balanced read/write performance, since updates do not involve neither index node nor memory layout reorganizations. Moreover, its deterministic behavior allows an efficient handling of parallel requests, because there are no costly internal reorganizations that depend on the actual data inside the index.

The contributions of this paper are the presentation of:

1. A fine-grained locking scheme and an efficient memory management subsystem for the generalized prefix tree as index structure, which allows non-blocking reads and single-lock writes. We discuss this locking scheme in Section 2 in more detail.
2. A complementary coalesced cyclic log [7, 6, 5] that is used to log each manipulating operation and to recover the in-memory index in case of hardware or software failure. This log that we describe in Section 3 is tuned to cooperate best with our in memory index structure.

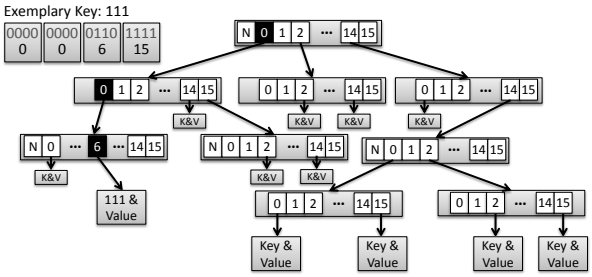


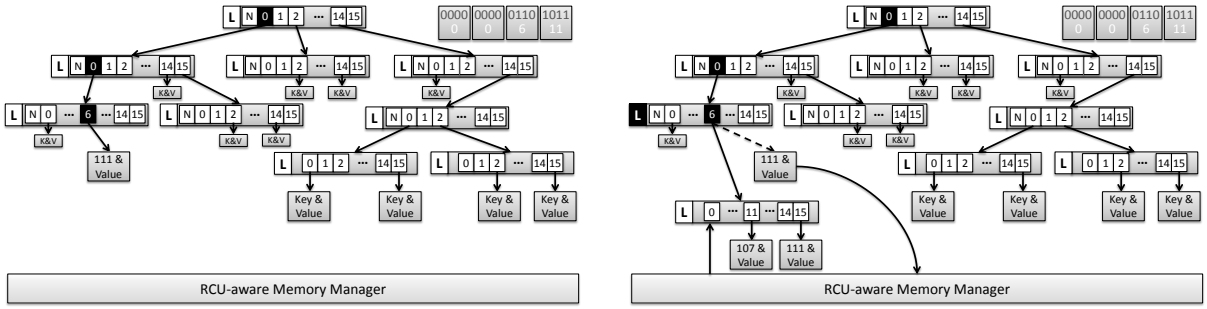
Figure 2: The Generalized Prefix Tree.

After dealing with both system parts, we evaluate each part as well as the overall indexing system in Section 4 on different hardware configurations. Finally, we conclude the paper in Section 5.

2. HIGH-THROUGHPUT IN-MEMORY INDEX

In this section, we start giving an introduction to the generalized prefix tree, followed by the additional changes we made to enable it to handle massive parallel requests. Figure 2 shows an example of a prefix tree in which we highlighted the traversal path for the 16 bit width key 111 (decimal notation). To find a key inside this prefix tree, the key is split into fragments of an equal prefix length k'. Starting from the left, each fragment is used to identify the bucket in the corresponding tree node. For example, the first four bits in this example are used to find the appropriate bucket in the root node for this key. This bucket contains a pointer to the next node that takes the next four bit fragment to find its bucket on this tree level. The number of buckets in each node depends on the prefix length k' and is calculated by $2^{k'}$. At the end of this traversal path is the actual content node, which contains the value for the searched key. So, the main character of a prefix tree is that the key itself is the actual path inside the prefix tree and is independent of other keys present in the index.

The most important configuration parameter is the static prefix length k'. For instance, a 16 bit key ($k = 16$), $k' = 1$ would cause a maximum tree height h of 16 which also leads to 16 costly random



(a) Tree Traversal.

(b) Node Split.

Figure 3: Example of how to insert a new Key into a Prefix Tree.

memory accesses. This configuration is similar to a classical binary tree. The other extreme is $k' = 16$, where only one huge node with 2^{16} buckets is created. Here, we have only one memory access to find a key's value at the cost of a bad memory utilization. For the contest, we set $k' = 4$ to fulfill the memory limitation on the one hand and the performance requirements on the other.

In addition to the base index structure, we applied some performance and memory optimizations as shown in [3]. The *dynamic expansion* expands a node only when a second key with the same prefix is inserted. The example in Figure 2 contains such a case. Key 111's content node is already linked in the third level of the tree. This is possible, because there is no other key inside the tree that uses the same prefix after this point. As soon as a key is inserted that shares the same 12 bit prefix but differs in the fourth fragment, a new node is created at the fourth level. The second optimization we applied is to store the type of the next node directly inside the pointer to that node. So, the highest bit of each pointer (8 Byte aligned memory) determines whether the next node is a content node or an internal node. This reduces the index size and the number of failed speculative execution steps, because the code path is determined much earlier.

The most challenging issue on modern hardware is parallelization. Thus, we are forced to identify a fine-grained locking scheme or even better, to use no locks at all, which is extremely difficult to design and in some cases not even possible. For our solution, we designed a locking scheme that allows non-blocking reads and write operations requiring only a single lock. However, the main bottleneck for the write performance is given through the SSD latency. We will show in Section 3 that due to the special characteristics of the flash-optimized log and the on-the-fly garbage collection, also write operations benefit dramatically from a high degree of parallelism.

In the first place, we describe how to protect write operations against each other. This is achieved by adding a single lock to each internal node. We decided to use spinlocks as the specific locking mechanism, because they (1) have less overhead than mutexes/futexes in their lock and unlock operations and they (2) occupy only 4 Byte of memory, which is much less compared to the size of a mutex structure, which is nearly as big as an entire cache line and therefore doubles the size of each node. Due to the deterministic behavior of our prefix tree, a lookup for a specific key takes always the same path inside the tree and—most importantly—there are no balancing tasks inside and between the internal nodes of the tree. Hence, the nature of the prefix tree allows us to perform a write operation by only locking a single node, because we do not have to lock across multiple nodes for, e.g., balancing purposes. Therefore it is enough to lock the node that needs to be split or where a content node has to be updated. A more fine-grained solution would be to lock single buckets instead of complete nodes, since this would require about 50% more memory for a node, we decided against this solution.

In order to allow non-blocking read operations, we use the read-copy update (RCU) mechanism [9]. With RCU, a content node is never updated in-place by just overwriting the old value with the new one, but it copies the current content node and modifies this new private copy. In a second step, the pointer, which referenced the old content node, is updated to point to the new content node. This allows readers that still read from the old version to finish and takes subsequent readers to the new version of the content node. A problem that arises with RCU is that the memory management needs to detect,

whether it is safe to recycle the old content node's memory block. We accomplished this by adding a counter to each content node that is atomically increased by a reader when starting to read from this content node and is atomically decreased when finished reading the content node. Thus, the RCU-aware memory manager has to test this field for zero, before it can be recycled. The memory manager itself is completely implemented in userland, because `malloc` calls turned out to be much too expensive. Therefore, the memory manager allocates one huge memory chunk via a `mmap` call at the beginning and administrates this chunk on its own. For memory recycling, the memory manager maintains a free list for each possible chunk length, which is limited through the maximum key and value sizes defined by the contest.

EXAMPLE 1. *To summarize, we provide an example in Figure 3. The example shows the write operation of the decimal key 107 (binary representation and fragmentation in the upper right corner of the figure). Compared to Figure 2, every internal node is now extended with a lock. At first, the running thread traverses the prefix tree down to the third level as shown in Figure 3(a). The thread now faces a situation in which the bucket is already occupied by a another content node with another key. Thus, it has to perform a* dynamic expansion *as shown in Figure 3(b). Therefore, it locks the internal node and checks whether the situation is still the same, otherwise it has to retry. At this point it is safe for the thread to work on that internal node. In the next step, the thread asks the memory manager to allocate the new node for the fourth tree level and two new content nodes. The value of the old content node is copied to the new content node and the key tail of the old node is truncated and written to the new one. The content node for the new node is created as usual and both new content nodes are linked by the new internal node. Now, the pointer of the third level node is turned to the new internal node and the node can be unlocked. In a last step, the thread returns the old content block to the memory manager, which is going to recycle its memory as soon as no reader is reading its memory anymore.*

3. COALESCED CYCLIC LOG

In this section, we present our flash-optimized coalesced cyclic log that is tuned to operate hand in hand with the in-memory index structure as depicted in Figure 1. A flash-based SSD basically consists of some flashpacks and a controller. The controller mainly implements the error correction and the wear leveling, which is responsible for prolonging the service life of the flashpacks that are the actual storage media. Main advantages compared to hard disks are better energy-efficiency, energy-proportionality, higher throughput, and lower latency what shortens the gap between transient main-memory and persistent drives. Previous research [4] showed that SSDs expose their full performance when reading or writing to it with a sequential access pattern, similar to hard disks and main memory, because flashpacks are not capable of doing in-place updates. Instead, a SSD has to erase a flash block of typically 4 KB first, before it is able to write this entire block again. Furthermore, flash memory is only able to erase a set of blocks (the erase block size between 128 and 512 KB) at once. All these internal characteristics are hidden from the user through the Flash Translation Layer (FTL). In order to exploit the full performance, we need to be aware of these internal limitations.

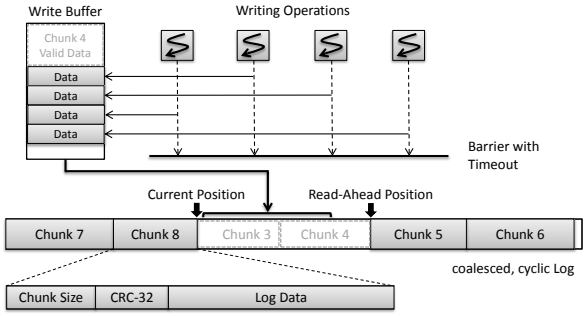


Figure 4: The coalesced cyclic Log.

Since the SSD I/O is the bottleneck of the complete indexing system, it is essential to write with a sequential pattern to the device. Thus, we decided to use an append log as base structure and applied the following two extensions:

1. *Write coalescing* to maximize the write throughput.
2. *Cyclic writing*, because of the limited SSD space.

The idea of *write coalescing* is similar to a group commit. Instead of writing each log record individually, we collect as much as possible log records from the simultaneously running operations in a write buffer and flush them at once. The *write coalescing* increases the overall throughput dramatically, because the contest demanded hard durability, which is ensured by drive cache flushes that are very costly operations with a high latency. As a side effect, this raises the latency of single writing operations. However, it is a good trade-off when taking the throughput gain into account. Figure 4 shows a schematic

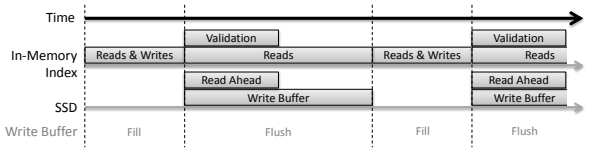


Figure 5: In-Memory Index and SSD Usage over Time.

Operation Type	Probability
Read	45%
Write	40%
Delete	5%
Compare-and-Swap	5%
Scan (max. 10 rows)	5%

Table 1: Distribution of Operation Types.

overview of our coalesced cyclic log. The central component is the described write buffer that collects the single log records from the individual threads. After a thread has written a log record to the write buffer, this thread is stalled until the write buffer is flushed. We flush the write buffer, either it is full, there is no other write operation left, or a predefined timeout is reached. Every time a flush is initiated, the single log records in the write buffer are composed to a chunk that is 4 KB aligned and checksummed with a CRC-32. Afterwards, this chunk is written to the disk using the `fdatasync` system call in Linux.

Since we do not write out the entire index structure as a checkpoint and the available space on the SSD is limited, the log needs to be *cyclic*ally overwritten. This forces us to perform a garbage collection on-the-fly. Thus, the coalesced cyclic log reads at least the size of the write buffer ahead. While reading, it validates the read log records against the in-memory index structure. All log records that are still valid are stored at the beginning of the write buffer and are written together with the new log records on the next write buffer flush.

Figure 5 shows the typical write buffer period and the activity states of the in-memory index as well as of the SSD. At first, when the write buffer is in the *Fill* mode, read and write operations are processed. All changes made by write operations are stored as log data in the write buffer. The corresponding threads are blocked until the write buffer is flushed. As soon as one of the *Flush* conditions for the write buffer occur, the write buffer is locked disabling further write operations. During the write buffer is flushed to the SSD, the storage system already reads ahead the log to free up the space for the next data chunk. The SSD's write performance is not affected by the simultaneous reading, because the operating system usually detects sequential read patterns and prefetches this data in the I/O buffer. After the data was successfully written to the SSD, the write buffer changes back into the *Fill* mode. In order to fully utilize the SSD, it is necessary to keep the time of *Fill* phases as small as possible, because the SSD becomes idle during these times. Thus, we spent a lot of efforts to allow fast parallel operations and non-blocking reads on the in-memory index. Another solution is to use two write buffers and alternately filling and flushing them. However, this solution turned out to be much slower, because of the high latency of a SSD flush operation. Another reason for making reads non-blocking is, that reads are allowed at any time and they are extensively used for log data validation. The main reason for using non-blocking reads was given through the overall scenario: In the contest, the benchmark was setup to create a specific amount of threads. Each thread has a given probability to issue either a read or a write operation and the storage system works optimally when all of these threads flush their writes at once. Therefore, we need to process read operations fast to have every thread doing a write operation to fill the write buffer as fast as possible. This finally prevents the SSD from being idle.

Once, the system crashes or is shutdown, the storage system must be capable of rebuilding the in-memory index. This is done by reading the log twice. The first time, we only process update operations and the second time we apply delete operations. To maintain the temporal order, the in-memory index as well as each log record contains sequential transaction numbers. Thus, an update respectively a delete is only applied, if the transaction number is greater than the current one in the content node of the in-memory index. The need for applying delete operations in a separate run results from this comparison.

4. EVALUATION

In this section, we evaluate individual system components as well as the overall performance on different hardware configurations and parameter settings. The evaluation system, which is different from the system used for the contest, is equipped with an *Intel i7-3960X (6 cores with Hyper-Threading, running at 3.3 GHz and 3.9 GHz max. Turbo Frequency, four memory channels and 15 MB shared L3 cache), 32GB of DDR3-1600, and an Intel X25-E 64GB SSD.* For benchmarking, we

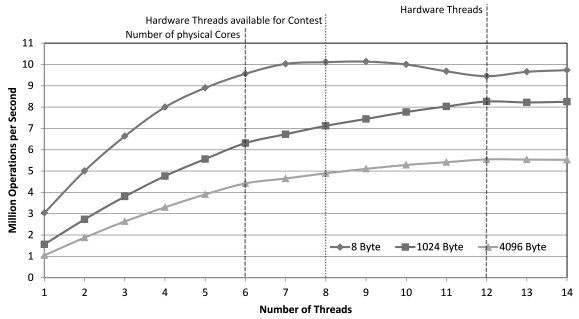


Figure 6: Pure In-Memory Index Performance dependent on the Number of Threads.

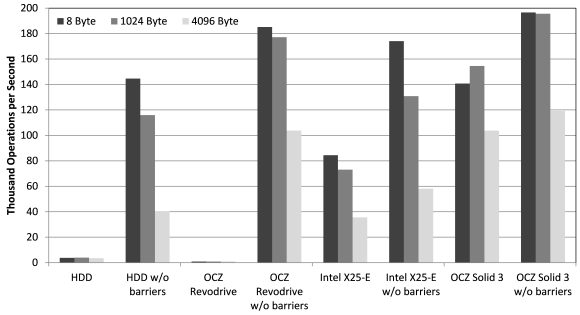


Figure 7: Overall Performance on different Drive Configurations.

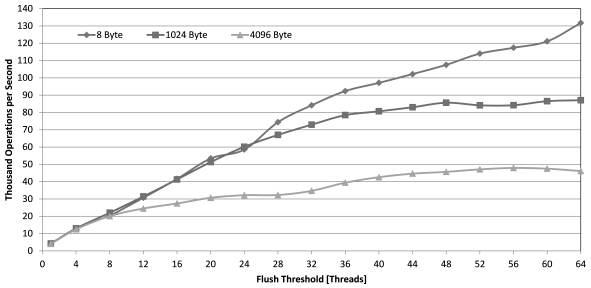


Figure 8: Overall Performance as a Function of the Flush Threshold.

used the benchmark driver provided by the programming contest. This driver generates 8 Byte sequential integer keys to populate the index structure. After that, the driver launches a given set of threads (32 as default) and each of them queries the index for a preset amount of time. Table 1 shows the probabilities for a thread to select a specific type of operation for the next query.

In the first experiment, we evaluate the performance of the pure in-memory index structure. Therefore, we completely disable the SSD log. Figure 6 presents the measurements in million operations per second for a range of 1 million sequential keys (uniformly selected from this range) with a payload of 8, 1024, and 4096 Bytes as values. Further, we marked some points specific to the evaluation hardware. For large payloads like 1024 and 4096 Bytes, we observe optimal scalability of the index. With up to 6 threads, where each of them can be mapped to an exclusive physical core, the performance scales nearly linearly. In the range from 6 to 12 threads, the cores are shared by two threads to fully utilize its processing units, the index still scales nearly linear, but with less gain. The performance gain in this region mainly depends on the remaining amount of memory bandwidth. After the limit of 12 hardware threads is reached, the performance gain stalls and starts to decrease, because of the scheduling overhead. When looking at smaller payloads like 8 Byte, we see another behavior. Here, the performance does not scale linearly, instead, the performance benefit of adding a new thread decreases constantly and even starts to lower the overall performance after nine threads until it reaches the hardware thread limit. This happens because the index is not memory bound anymore and is now facing the high concurrency overhead, especially in the memory management subsystem, when writing a key/value pair to the index. Due to the fact, that the evaluation machine of the contest only got 8 hardware threads available, we did not have any performance penalty by allowing more than eight thread to work in parallel.

In the second experiment, we activated the SSD log and measured the index performance for four different drives. We used an PCI Express OCZ Revodrive, which internally consists of two SSDs connected via a RAID-0 chipset, an Intel X25-E 64GB SATAII enterprise class SSD, and a mainstream OCZ Solid 3 64GB SATA III SSD. Moreover, we tested our solution on a classic Samsung 160 GB SATAII HDD to compare the results with the SSDs. For each drive, we measured the performance for three different payload sizes with and without ext4 barriers. An ext4 barrier guarantees the drive cache to be flushed when calling `fdatasync` (assuming the driver controller supports the `FLUSH CACHE` command). With disabled ext4 barriers, only the file cache is written back to the drives cache. In case of a power loss, the data is not guaranteed to be on the drive. Figure 7 shows all results. In general, we observe major differences between the different drive types with enabled barriers. The worst performance was measured on the Revodrive and the HDD, which are outperformed by orders of magnitude by the X25-E and the Solid 3 drive. With this setting, we mainly measured the latency of the con-

troller respectively the mechanical movements on the HDD, because there is no way of hiding the latency with the drive cache anymore. When turning off the ext4 barriers, we are able to measure the disks bandwidth. Here, the HDD performs much better, because HDD controllers are tuned for latency hiding. There are not much expensive mechanical seeks necessary, because of the sequential write pattern. Furthermore, a HDD is able to overwrite sectors without erasing or copying them first. The best results are achieved with the Revodrive and the Solid 3, which is mainly dedicated to fast PCIe x4 respectively SATAIII interface.

The last experiment, demonstrates the impact of the coalesced writes. This experiment was executed by 64 threads in parallel on an Intel X25-E for different payload sizes and write cache thresholds. For example, a write cache threshold of 4 means that the write cache is immediately flushed after it collected 4 single log records. The respective measurements are visualized in Figure 8. As an overall result, we see that the total performance benefits massively from a high threshold configuration, because the cache flush is the actual bottleneck in the system. When comparing the results for the different payload sizes, we observe that the benefit of flushing more writes at once decreases earlier for big payloads than for the small ones, what can be explained with the SSD hitting its bandwidth limit when transferring the data from the write buffer in the main memory to the drives cache, before it is able to flush this cache.

5. CONCLUSION

With the wide availability of large main memory capacities and multi-core systems, in-memory indexes with efficient parallel access mechanisms become more and more important to databases. Application trends on the other hand, demand hard durability requirements and high update rates, which can not be sustained by classic B-Tree like index structures on conventional hard disks.

Our solution of the SIGMOD Programming Contest 2011, that we presented in this paper, addresses exactly these issues. We designed an indexing structure with an efficient locking scheme that scales with the growing number of hardware threads and exhibits a balanced read/write performance. This structure is based on the generalized prefix tree, which we augmented with an efficient locking scheme to allow non-blocking reads and single-lock writes for fast parallel access. To fulfill durability requirements, we built a storage system that incorporates into this indexing structure and takes advantage of the characteristics of modern flash-based SSDs. The storage system is mainly a cyclic log that collects single log records in a write buffer before flushing it to disk to achieve maximum throughput. The orchestration of both components — the index structure and the storage system — creates a powerful indexing system for modern applications.

6. ACKNOWLEDGMENTS

We thank the NSF, Microsoft, and the ACM for sponsoring this contest and especially the MIT CSAIL for doing such a great job in organizing it. Furthermore, we thank all the other participants for pushing each others solutions forward. This work is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

7. REFERENCES

[1] SIGMOD Programming Contest 2011. `http://db.csail.mit.edu/sigmod11contest/`.

[2] R. Bayer and E. McCreight. *Organization and Maintenance of Large Ordered Indexes*, pages 245–262. Software pioneers, New York, NY, USA, 2002.

[3] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, pages 227–246, 2011.

[4] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[5] S. Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD*, pages 73–86, 2009.

[6] B. K. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. *PVLDB*, 3(2):1414–1425, 2010.

[7] B. K. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *SIGMOD*, pages 25–36, 2011.

[8] P. L. Lehman and s. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.*, 6:650–670, December 1981.

[9] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems.

[10] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. *SIGMOD Rec.*, 29:475–486, May 2000.