Till Kolditz, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Wolfgang Lehner

Online bit flip detection for in-memory B-trees on unreliable hardware

Erstveröffentlichung in / First published in:

SIGMOD/PODS'14: International Conference on Management of Data, Snowbird 23.06.2014. ACM Digital Library, Art. Nr. 5. ISBN 978-1-4503-2971-2.

DOI: https://doi.org/10.1145/2619228.2619233

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

Online Bit Flip Detection for In-Memory B-Trees on Unreliable Hardware

Till Kolditz, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner

Database Technology Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Hardware vendors constantly decrease the feature sizes of integrated circuits to obtain better performance and energy efficiency. Due to cosmic rays, low voltage or heat dissipation, hardware – both processors and memory – becomes more and more unreliable as the error rate increases. From a database perspective bit flip errors in main memory will become a major challenge for modern in-memory database systems, which keep all their enterprise data in volatile, unreliable main memory. Although existing hardware error control techniques like ECC-DRAM are able to detect and correct memory errors, their detection and correction capabilities are limited. Moreover, hardware error correction faces major drawbacks in terms of acquisition costs, additional memory utilization, and latency. In this paper, we argue that slightly increasing data redundancy at the right places by incorporating context knowledge already increases error detection significantly. We use the B-Tree – as a widespread index structure – as an example and propose various techniques for online error detection and thus increase its overall reliability. In our experiments, we found that our techniques can detect more errors in less time on commodity hardware compared to non-resilient B-Trees running in an ECC-DRAM environment. Our techniques can further be easily adapted for other data structures and are a first step in the direction of resilient database systems which can cope with unreliable hardware.

1. INTRODUCTION

For the last 30 years, disk-centric database systems have been state-of-the art. Within the recent years, this approach has dramatically changed due to several reasons. Especially the significant developments in the hardware sector are the major driver for that change. Due to these hardware developments, servers with 2 TiB of main memory are available for a reasonable price. To efficiently leverage the provided main memory capacities, the database system architecture trend shifted from a disk-centric to a main memory-centric approach, where the entire data pool is kept completely in main memory. The performance gain is massive because database operations are able to benefit from its higher bandwidth and lower latency [9].

The driving hardware developments mainly rely on shrinking the feature sizes of integrated circuits forming processors and main memory. On the one hand, this hardware trend in combination with appropriate software concepts are responsible for performance improvements, in particular in the database domain. On the other hand, hardware becomes more and more vulnerable to external influences such as cosmic rays, electromagnetic radiation, low voltages, and increased heat dissipation. Besides manufacturing errors, hardware also takes permanent damage over time which is known as aging. Especially main memory already faces those effects resulting in an increased bit error rate, which is likely to escalate on future DRAM hardware. From a database perspective, an increased bit flip rate in main memory will become a major challenge for modern in-memory database systems, which keep all business-related data in volatile and unreliable main memory.

The challenge of DRAM reliability is not new and for years manufacturers established ECC-DRAM which transparently detects and corrects bit errors. Common ECC-DRAM protects against single bit DRAM errors and provides detection for double bit errors using a 72/64 Hamming code. This basic hardware protection requires 12.5% more memory cells per bit. Newer hardware approaches use more sophisticated methods to correct and detect multiple bit flips, whereas memory and computation overhead increase. However, this hardware approach is not only limited in its error detection and correction capabilities, it also faces major drawbacks in terms of acquisition costs and memory latency as illustrated in Table 1. There are no DDR3 (registered) ECC modules with CAS latency (CL) below 9, which incurs a 29% latency penalty compared to CL7 non-ECC DDR3 modules at almost the same price – CAS is only one DRAM timing parameter and other timings are accordingly higher as well.

Aside from hardware solutions for reliability, there also exist other approaches like redundant data storage and redundant computation [18]. In this case, most errors are detectable by comparing the final results with each other. However, the major problem of ensuring a low error probability by employing generally applicable methods is dramatically increasing costs for memory and computational power. In this paper, we argue that adding a varying degree of data

DOI: http://dx.doi.org/10.1145/2619228.2619233.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

redundancy by incorporating database context knowledge is able to reduce those costs. More specifically, we concentrate on online bit flip detection for in-memory B-Trees and are thus taking the first step towards bit flip-tolerant database systems. Our approach leverages the knowledge about the internal structure of the B-Tree to efficiently detect bit errors on a logical level and adds different degrees of redundancy to scale with the amount of bit errors generated by DRAM hardware and the level of reliability that is requested by the application. Moreover, we compare lightweight and heavyweight error detection mechanisms with regard to their impact on performance and main memory utilization. In our evaluation, we found that our techniques can detect more errors on commodity hardware compared to a non-resilient B-Tree running in an ECC-DRAM environment that comes with several drawbacks in terms of increased TCOs and lower performance.

Contributions. Our contributions can be summarized as follows:

(1) We present the results of a basic experiment that shows the increasing bit error rate and corresponding bit error distribution when operating DRAM outside of the normal operating parameters (in particular overheating).
(2) We describe different bit error detection techniques for non-resilient in-memory B-Trees that leverage the knowledge about the structure of a B-Tree and tolerate a specific error rate.
(3) We evaluate our bit error detection techniques and quantify their impact on performance as well as on memory consumption. The results show that we are able to detect more bit errors at higher performance and with lower memory footprint compared to standard B-Trees running on an ECC-DRAM system.
(4) Based on our findings, we point out an important research direction that uses an adaptive degree of redundancy for database structures to scale with DRAM error rates and application demands.

Outline. The paper is structured as follows: In Section 2, we present our basic experiments that show increased bit error rates by overheating DRAM. Then, we briefly review related work in Section 3. Afterwards, we introduce our developed Error Detecting B-Tree (EDB-Trree) variants in Section 4. We start with a short description of our baseline B-Tree implementation, followed by different error detecting variants. In Section 5, we evaluate the impact of our EDB-Tree variants on performance and main memory overhead. Finally, we close the paper with a conclusion in Section 6.

2. MAIN MEMORY ERROR RATE EXPERIMENTS

Several researchers have already pointed out that main memory becomes a severe cause for hardware based failures in today's data centers [11, 19]. In general, there are some common causes for memory errors, each showing different characteristics. As described in [2], we are able to distinguish between static or hard errors as permanently corrupted bits and dynamic or soft errors as transiently corrupted bits. In particular, dynamic errors are produced by cosmic rays, electromagnetic radiation, low voltages, and increased heat dissipation. The impact of cosmic rays was already investigated several years ago and experiments showed that smaller feature sizes tend to be more vulnerable to cosmic radiation [3, 17]. Next to cosmic rays, heat is another factor known to cause dynamic error in DRAM [15]. Therefore, we provide an experiment showing the correlation between operating temperature and DRAM error rate as well as the corresponding error distribution. Afterwards, we draw some implications for main memory database systems.

CL	Latency	Type	Costs [€]	Rel. Costs
7	1.00	non-ECC	83.79	—
8	1.14	non-ECC	66.94	—
9	1.29	non-ECC	62.32	1.00
		ECC	—	—
		reg ECC	81.65	1.31
11	1.57	non-ECC	60.49	1.00
		ECC	74.13	1.23
		reg-ECC	69.99	1.16
13	1.86	non-ECC	67.49	1.00
		ECC	87.62	1.30
		reg-ECC	125.79	1.86

Table 1: Comparison of single module DRAM and ECC-DRAM with the costs taken from a German price comparison web site on May 26th, 2014. The costs are averaged over the 10 cheapest modules (where available). Rel. = Relative, according to CL-category

2.1 Heating Experiment

To understand the implications of heat induced memory errors (i.e., the distribution of bit flips), we conducted an experiment for different DRAM operating temperatures. For the measurement setup, we used a heat gun to operate the memory at several temperatures. Moreover, we used an infrared thermometer to measure the current temperature of the DRAM. To evaluate the distribution of heat induced bit flips, we used a software tool that continuously scans a large memory area for bit errors. Therefore, this tool allocates a large portion of memory as an 8-Byte integer array in a continuous physical memory range (we directly mapped the `/dev/mem` device into the virtual address space) and writes to each position in the array the corresponding array index or zero. Consequently, the content can be easily validated for corruption respectively bit flips by comparing each 8-Byte value with its array index or zero. Due to the continuous allocation of the physical memory, we are able to estimate the error distribution in the DRAM under different heating circumstances.

To precisely evaluate the influence of heating, we conducted multiple runs in an experiment, whereas a single run scans the whole array in an ascending order from the first to the last index and logs detected errors in a file on disk. The heating curve over the multiple runs corresponds to a Gaussian curve, that means, we increased the induced heat from the beginning until we reached a maximal temperature. Then, the remaining runs are executed under diminishing temperatures. Furthermore, we performed experiments in two different scenarios: (1) only detecting errors and (2) with correcting corrupted values by directly overwriting them the the right predefined value (index position or zero). These scenarios are necessary to evaluate the bit

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

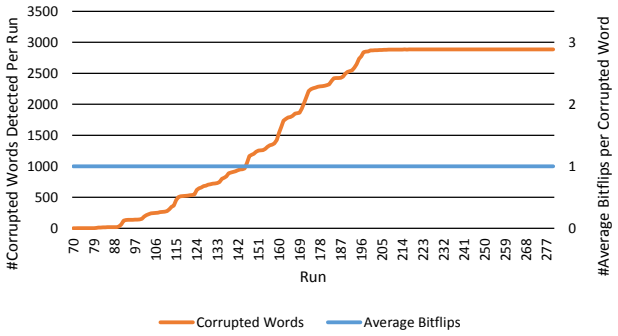

Figure 1: Heating Experiment without Correction based on allocated memory filled with zero values (using new main memory modules).

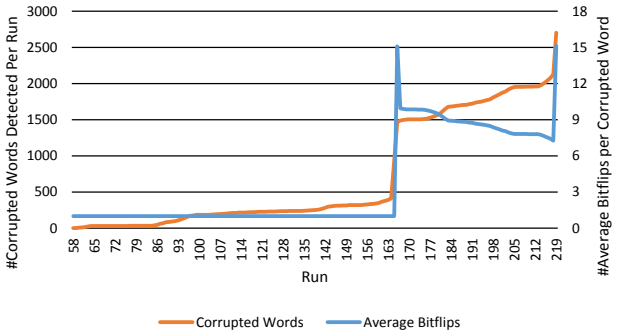

Figure 2: Heating Experiment without Correction based on allocated memory filled with index values (using older and commonly used main memory modules).

flip rate without and with correction.

Our test system was a Pentium D clocked at 3 GHz with 2GiB of DDR2 main memory running an Ubuntu Linux as operating system. The tool was instructed to allocate an array of 1.5GiB so that no swapping of memory occurs. Regarding performance, the tool scanned that amount of memory 2.6 times a second. While running the program, we tried to heat up the memory modules uniformly and observed the temperature with the infrared thermometer. Forcing memory errors through heat worked reliably in our setup when the memory chips reached a temperature of 100 °C and above. The actual observations regarding our two different scenarios are as follows:

Scenario 1: Experiment without Error Correction

In our first experiment, we conducted our multiple runs without error correction. Our expectation of this experiment is, that the number of corrupted 8-byte words increases with increasing heat. Figure 1 shows the results of this experiment, based on an allocated array filled with zero values. As we can see, the total number of corrupted 8-byte words increases in an accumulated manner with almost each run as expected. In order to reduce the number of corrupted 8-byte words over the runs, we need to correct the detected errors in each run as explained later on.

A further observation of this experiment is, that the average bit flip rate per 8-byte word is constant by 1. That means, the number of bit flips is very low which could be efficiently handled by simple ECC-DRAM. However, an important notice is that the utilized DDR2 memory modules were new while conducting this experiment. We utilized the main memory modules further on to include the aging aspect in our experiment. Figure 2 shows results of a further experiment later on. In this case, the allocated array was filled with corresponding array index positions. Again, we conducted several runs with increasing and decreasing heat. The results are almost the same as highlighted in the previous experiment. Nevertheless, we observe a new aspect. The average number of bit flips increases to a maximum value of 15 per 8-byte word. Therefore, the detection as well correction of words with a high number of bit flips is challenging.

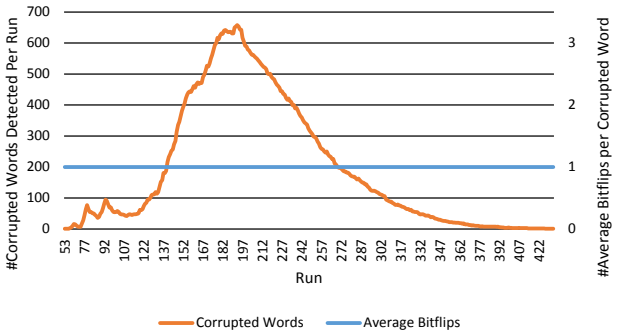

Figure 3: Heating Experiment with Correction based on allocated memory filled with index values (using new main memory modules).

Scenario 2: Experiment with Error Correction

In the second scenario, we slightly modified our experiment setting to investigate the influence of error correction. Figure 3 shows the result for an experiment with array index position. Again, we conducted multiple scan runs over the array with increasing as first and decreasing heat as second. During the scan runs, we detect errors and correct the detected errors in online fashion. As we can in the figure, the number of corrupted 8-byte words in each is lesser with correction as expected. Moreover, the diagram shows, that the number of corrupted words is correlated with the operating temperature. Furthermore, the average bit flip rate with new memory modules is constant by 1. With older modules, the characteristic changes and the bit flip rate increases as well as the total number of corrupted pages as depicted in Figure 4.

2.2 Implications

Based on the previous presented experimental results, we are able to conclude the following aspects for the main memory database community:

1. With managing all business relevant data in main memory, we have to cope with increasing errors and bit flip rates due to several reasons (heat, cosmic rays, electromagnetic radiation, low voltages).

2. In order to handle increasing errors, we require efficient techniques to detect as well as to correct these errors in

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

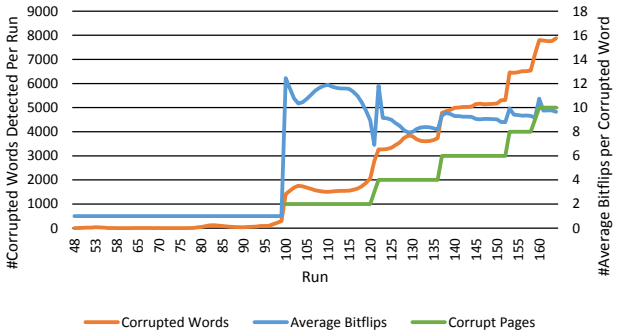

Figure 4: Heating Experiment with Correction based on allocated memory filled with index values (using older and commonly used main memory modules).

an online fashion. Whenever we access data, we have to validate the data. If the data is corrupt, the data has to be corrected immediately.

3. Furthermore, we require a set of techniques to cover all possible situations. Data that is accessed less frequently (cold data) require more sophisticated techniques compared to hot data that is accessed frequently and thus is corrected in shorter periods. In the case of cold data, errors are able to accumulate which complicates the error detection.

3. RELATED WORK

Fundamentally, for fault tolerance against dynamic memory errors several techniques are known. First, a general applicable approach is executing the same computation multiple times [18]. In this case, any dynamic error can be detected by comparing the final results. The most well-known technique in this class is triple modular redundancy. Second, error detection and error correction coding, such as parity and Hamming codes, introduce information redundancy to data [8]. Regarding DRAM bit flips the most commonly used approach is hardware-based (72,64)-Hamming ECC [16]. It realizes single-error correction and double-error detection. Many other general coding algorithms are available such as cyclic-redundancy codes or Reed-Solomon codes [16]. These enhanced coding schemes are more robust against burst errors than parity or Hamming, however their coding results in higher computational costs.

As already stated in [2], an increasing dynamic error rate with increasing costs for general-purpose error detection and correction codes as described previously, data management system should use their context knowledge for more efficient and more effective fault tolerance mechanisms. There exist already several works that discuss resilient data structures. Finocchi et al. [6] proposes resilient search trees that allow search, insert, and delete operations. The key idea of their approach is to form groups of keys and basically store for each group, the interval in which the group's assigned keys reside. The interval information allows to partially restore corrupted values as well as operations on non-corrupted values. There are also similar works, which provide resilient linked-lists [1], priority queues [12], or dictionaries [4]. All of them exploit additional redundancy in their data structures to detect errors and cope with them. However, to the best of our knowledge, besides errors due to disc problems [10] there is no work so far on making data structures heavily used in database systems – like the B-tree – resilient against arbitrary memory corruption.

Besides data structures that can cope with errors, there exist further many fault-tolerant algorithms. Finocchi et al. [7], for example, propose a fault-tolerant sort and binary-search algorithm. Both algorithms can cope with corrupted keys and deliver in most cases correct results for the uncorrupted values, i.e., the uncorrupted values are correctly sorted within the sort algorithm and correct uncorrupted keys are returned when searched within binary search. The basic idea of these approaches is that they perform redundant work to cope with corruptions, i.e., additional checks are applied which would not be performed in the original algorithms.

4. ERROR DETECTING B-TREES

In this paper, we concentrate on *online bit flip detection* for in-memory B-Trees and are thus taking the first step towards bit flip-tolerant database systems. B-Trees [5] are widely used in database systems to accelerate point and range queries and therefore they are an important component of database systems. To show the benefits of implementing appropriate error detection mechanisms leveraging context knowledge – in contrast to general-purpose mechanisms – we present different adaptations of the B-Tree to detect bit flips in main memory on the fly. Since there exists a wide range of B-Tree implementations, we adapt a basic B-Tree without any special performance tweaks in this paper. First, we will present our basic B-Tree implementation and the widely used "naive" triple modular redundancy (TMR) variant for error tolerance, which are used as baselines in our evaluation. Second, we propose three different lightweight to heavyweight adaptations for detecting bit flips in main memory B-Trees.

The baseline B-Tree and all variants are designed for single-threaded applications, as the main focus lies on investigating mechanisms for detecting arbitrary main memory bit flips. Anyhow, there are also DBMSs employing single-threaded query processing, like VoltDB [20] and H-Store [13].

4.1 Baseline Trees

B-Tree

The physical layout of our B-Tree serving as a baseline is shown in Figure 5, which shows 3 inter-linked nodes whereas black and red arrows denote links from a node to its children or from a node to its parent, respectively. Each node is structured the following. At first, there is a pointer P to the node's parent node which is *null* in the case of the root node. Then, there follows a 2-Byte integer L denoting the node's *level* inside the tree, which is 0 at the leaf level and increases by one at each higher level towards the root. Afterwards, another 2-Byte integer F (fill level) represents the number of key-value pairs currently contained in the node. At next, there are the key-value pairs $\{(K_1, V_1), (K_2, V_2), \dots\}$ where each key is 4 or 8 bytes wide as well as each value, which e.g. can be inline data, a RID (record identifier), or a pointer to the actual data. Inside a node, keys are kept in ascending order. Finally, there follow pointers to child nodes $\{P_1, P_2, \dots\}$, which are always 8 bytes long. Finally, a padding – not shown in the figure for conciseness – allows to align the node to a desired boundary – usually the system's memory

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

Figure 5: Physical layout of B-Tree and EDB-Tree (without checksums).

Key width [Bytes]	4	8	4	8
Value width [Bytes]	4	4	8	8
K (B-Tree)	127	101	84	
K (EDB-TreeCS)	126	101	84	

Table 2: K for different key and value sizes for baseline B-Tree and EDB-TreeCS.

page size of 4KiB. Note also that the compiler (G++) aligns L and F so that they together consume a total of 8 Bytes for better alignment of the subsequent node members.

As is common for B-Trees (and its derivatives) parameter K delimits the number of key-value pairs and pointers in nodes: except for the root, in a node there must be present $K \ldots 2K$ keys and accordingly up to $2K + 1$ child pointers. For 4 byte keys and values K is 127 (254 keys and 254 values per node), while for 8 byte keys and values K is 84 (168 each per node). Accordingly, there are up to 255 or 169 children for a node, respectively. For mixed data widths of 8 byte keys and 4 byte values or vice versa K is 101. Table 2 summarizes the values for K according to the data width of keys and values for our B-Trees.

This baseline employs no error detection at all. Corrupt pointers to unallocated memory result in segmentation faults, while corrupted keys and pointers to the wrong child may result in false positives or false negatives – keys which were never added to the tree are found and the other way around. For all variants presented hereafter methods for building the trees and doing point queries are fully implemented.

Triple Modular Redundant B-Tree

Triple Modular Redundancy (TMR) is a common technique for tolerating arbitrary data corruption. The basic idea is to keep three copies – replicas – of all data, execute algorithms on all three copies, compare the according results and finally do a majority voting to opt for the correct result. The TMR baseline instantiates three B-Trees replicas and performs each and every operation on all replicas successively, one after another. For instance, a point query is successively forwarded to each replica, each result is temporarily buffered – whether found or not found and if found also the associated value – and then compared them against each other. If there is a successful majority voting, i.e. two out of three are the same, that result indicates whether the searched key was found or not. Also, when the key was found, the actual values are compared against each other and must match.

TMR succeeds as long as two replicas yield the same result but does not guarantee that this result is really correct: When at least two replicas are corrupted in the very same way, the error is not detected. Furthermore, execution time and required amount of memory are tripled, which might not always be acceptable.

4.2 Error Detecting B-Trees

In order to cope with an increase in error rates in future hardware, our idea is to introduce *online* checking for main memory corruption – in the form of bit flips – during tree traversal and node scans. Since we propose to make B-Trees detect main memory errors, we call these variants Error Detecting B-Trees, or short EDB-Trees. Furthermore, our premise for the following is that main memory may become corrupt, whereas CPUs – cores and on-chip caches – are reliable components not introducing further errors to data or during computations.

EDB-Tree

As a first variant, we add pointer sanity checks, while the physical layout of the tree and its nodes is still equal to those of the baseline B-Tree similar to [14], which did not provide any performance evaluation. Different from [14], however, we exploit virtual memory management:

1. the virtual address space is much larger than typically employed amounts of main memory,
2. hardware supports less than 64 bits for addressing – today usually 46-48 bits – and
3. operating systems further reduce the available addressable amount of main memory to 43-41 bits, for Linux and Windows respectively.

For utilizing the much larger virtual address space, EDB-Trees allocate nodes at successive virtual addresses. Consequently, from the process's perspective, the tree consists of a single contiguous memory area. This memory allocation allows to detect corrupt pointers pointing out of this area into unallocated virtual memory space, since we know starting address and length of the contiguous memory area. Nowadays these bits which would be zero anyways are used for further status information, but with regard to arbitrary bit flips in main memory this information again must be guarded against corruption and specially handled. Thus, we decided against using those bits in pointers for other purposes in this paper.

During tree traversal, three different pointer sanity checks are employed with minimal computational overhead: alignment, memory region and parent-child-relation. Since the node size is fixed, pointers must be aligned accordingly, i.e. for 4KiB nodes, the first 12 bits must be zero. Knowing the memory area's offset and size, the pointer is then range-checked, i.e. it must point into the allocated region. For handling pointers to the wrong child, the child's parent pointer is compared against the node's address from which we descended. If any of those three checks fails, we can be sure some bit flip(s) occurred. While the alignment- and region-checks suggest an error in the checked pointer, the parent-child relation only indicates one of the two has changed.

These sanity checks imply no additional memory footprint. However, they can detect only some errors in pointers where the position of the flipped bit – starting from 1 at the least significant bit – is greater or equal than $ld(node\ size)$ and smaller or equal to $ld(allocated\ memory)$. For instance, such an EDB-Tree using 4KiB nodes and having allocated

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

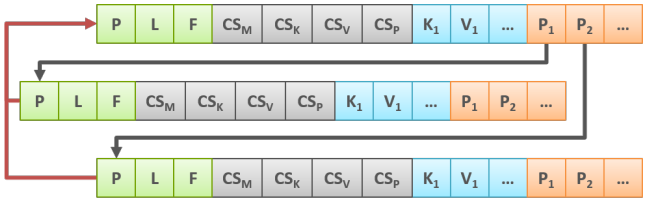

Figure 6: Physical layout of EDB-Tree with checksums.

4GiB of memory can not detect bit flips in pointers between bit positions 13 and 32, inclusive. Consequently, about 69% of bit flips could be detected in 64-bit pointers by those sanity checks.

The following variants all inherit the pointer sanity checks. Since main memory bit flips may occur at any position, any information added to nodes as some kind of redundancy for error correction must be guarded against main memory bit flips again. Therefore, the following techniques only add little to no redundancy – as e.g. in contrast to [14].

EDB-Tree Using Parity Bits (EDB-Tree PB)

For a given data word, a parity bit guarantees that there is always either an even or odd number of ones. By that, any odd number of bit flips can be detected, while even numbers of bits flip cancel each other out. Parities can be either even or odd and in principal there is no difference in employing either. Compared to e.g. 64 bits, a single bit causes an additional memory footprint of 1.6%, while for 32 bits it is 3.1%. Computing a parity bit on modern processors can be done with native instructions in a few cycles. Consequently, parity bits provide limited bit flip detection with very little memory and performance impact.

In this B-Tree variant, for each node member (compare Figure 5), a parity bit is computed and set as its most significant bit (MSB). Thus, for computing the parity bit, only all but the MSB are regarded and the according member's domain is reduced by one bit. For pointers, this is no limitation since the virtual address space is much larger anyways. The parity bit is set for a key and a value, whenever a new key/value-pair is added, or when a value is updated. It is set for a node's fill level whenever that changes, and on pointers when these change in the course of split, merge and delete operations. For each tree traversal, node members are first validated against the parity bit before their first use.

EDB-Tree Using Checksums (EDB-Tree CS)

For detecting more bit flips, we employ XOR checksums to groups of node members, because on the one hand these can be computed easily and quickly by XOR-ing the concerning data elements and on the other hand they will detect almost any bit flips. The exception is when exactly an even number of bit flips occurs at the very same bit position in an even number of data objects, so that these even each other out during XOR-ing of the values. An advantage of using XOR checksums is that they allow to correct a single data element, when there is only one corrupted element and when it is known exactly which element is the corrupt one. To improve this rate for a single node keys, values and pointers can be partitioned and one checksum for each partition can be added, which however further increases the memory overhead leading to a smaller fan-out. However, checksums like these introduce a greater performance penalty than parity bits since all according data elements used to compute the checksum must be accessed to verify the checksum again.

We add four checksums to each node: one for the parent node pointer, tree level and fill level (CS_M), one for all keys (CS_K), one for all values (CS_V) and one for all pointers (CS_P), as can be seen in Figure 6. This has a rather small impact on parameter K, as is depicted in Table 2: for 4-byte keys/values there remain 252 keys/values instead of 254, the memory overhead is 24 bytes, and the fan-out decreases by 0.8% while for 8-byte keys and values it is still 168, with an additional 32 bytes which fit into previously unused memory.

CS_K is updated when a key is added or removed, CS_V when value is added, removed or updated, CS_P when a pointer is added or removed (e.g. during a split), and CS_M when parent pointer, tree level or fill level change. During tree traversal, first CS_M is validated, while CS_K is validated while scanning through a node, i.e. while comparing the present keys with the searched one they are XOR-ed. Before descending or when the key is found all remaining keys are XOR-ed and the computed checksum is compared against the stored one. Also, only when the key is actually found in the node is the values checksum validated. Additionally, before descending CS_P is validated by XOR-ing all child pointers.

5. EVALUATION

In the following we show how our error detection techniques perform against ECC and TRM. Therefore, we first measured point-query throughput without bit flips to get a performance baseline. Then, we induced differently severe bit flip patterns ranging from 1 to 5 random bit flips per data word, to show how the presented techniques can detect more than 2 bit flips, in contrast to ECC. For this paper, the basic operations *insert(key, value)* and *get(key)* were implemented in order to build the trees and measure their throughput and bit flip detection capabilities. Results were obtained for 8-byte keys/values and 4KiB nodes on an ASUS P9X79 Pro mainboard running a 12-core Intel i7-3960X CPU and 8x4 GiB (32GiB) DRAM.

5.1 Throughput

Scanning a single node for the occurence of a key can be done in at least two different ways by iteratively scanning the whole node or by employing binary search. We tested both methods and received very different results, which are displayed in Figures 7 to 10. These show absolute and relative throughput without bit flips for tree sizes in numbers of keys ranging from 6.4 thousand up to 64 million to reflect very small and large trees. Actual sizes range from 32KiB to 1.4GiB from smallest to largest one, whereas for TMR those values triple. Memory overhead for checksum EDB-Trees was less than 3% in all cases. Each result was computed as an average over 10 runs.

For the iterative scan we employed loop unrolling for four elements. The results are shown in Figure 7 for absolute and in Figure 8 for relative performance. For a scan the baseline B-Tree reaches about 12.0M lookups for a very small tree of 6400 keys and 2.0M lookups for the largest one. The TMR baseline has a relative throughput of about 30%. Pointer checking of EDB-Tree incurs almost no performance penalty and even is slightly faster for a very small tree, which might be due to better prefetching, but that effect is nullified already for 64K keys. EDB-Tree PB however has a

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

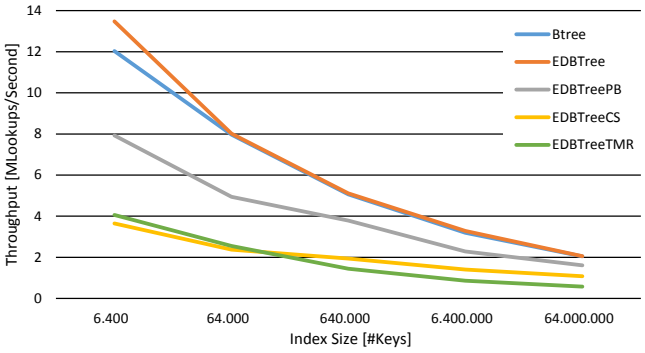

Figure 7: Absolute performance using iterative scan for searching keys.

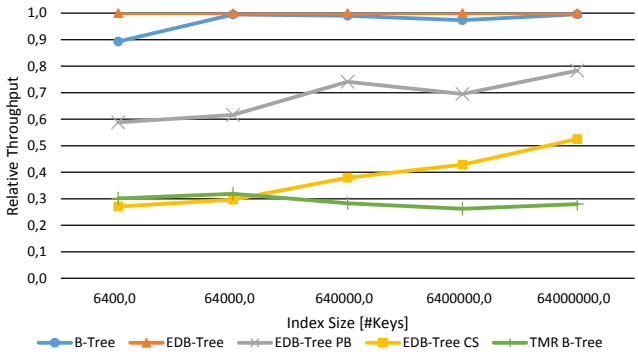

Figure 8: Relative performance using iterative scan for searching keys.

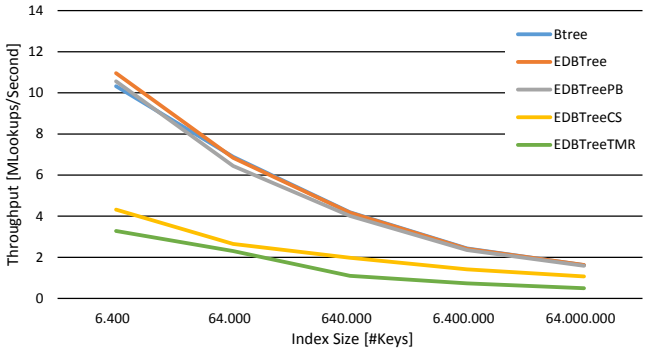

Figure 9: Absolute performance using binary search for searching keys.

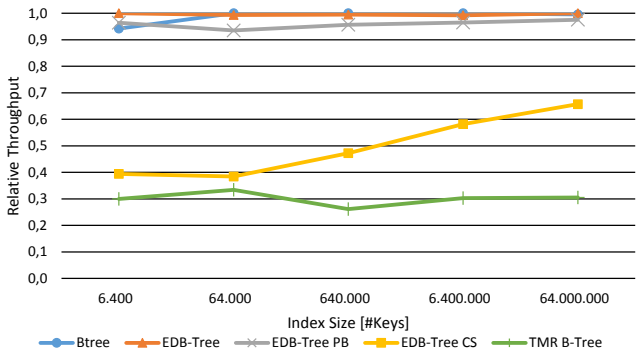

Figure 10: Relative performance using binary search for searching keys.

higher performance penalty of 40%...20% from the smallest to the largest tree, respectively. The checksums have an even greater impact of about 73% to 47%, respectively. As the larger trees become memory bound the additional computation overhead becomes less influencing.

When employing binary search the picture is quite different. Not only do the B-Tree and TMR variant have lower throughput, about 10.3M and 3.3M lookups per second, but the parity bit and checksum variants have higher performance for the smaller trees. The relative performance of the parity bit variant is almost as high as the B-Tree one's, because much less parity bits have to be evaluated. And even for EDB-Tree CS the relative and absolute performance improves, although the key checksum now is evaluated before the binary search. There the relative perfomance improves to 39% and 66% for the smallest and largest trees, respectively.

5.2 Bit Flip Detection

We examined five bit flip scenarios where 1 to 5 random bit flips per 8-byte word in the according tree memory ranges were simulated while there was a constant load of point queries on the trees on only the inserted keys. The 8-byte words for corruption were chosen uniformly across the trees' allocated memory and also the bit flip positions were chosen randomly. For each scenario and B-Tree variant over a total period of 300 seconds at every millisecond an 8-byte word was corrupted. For this experiment all trees used binary search as explained above. To measure the effectiveness of the techniques, we differ between two categories of errors. Detected errors are those which are detected by our proposed adaptations and for the TMR B-Tree when different results were encountered. Undetected errors are those which would otherwise not be identified by the techniques. To measure the second category, the trees were built with the values set to the according keys and only the inserted keys were queried. By that, false negatives – a key was not found although it must be present – and corrupted values – by comparing key and returned value – are detected. Whenever an error is discovered an exception is thrown indicating the type of error and also segmentation faults are caught and counted as undetected errors.

Figures 11 to 14 show how the tree variants were able or unable to detect 4 and 5 bit flips per 8-byte word, respectively. EDB-Tree PB and EDB-Tree CS are able to detect much more errors for 5 bit flips – about 73K and 168K errors during the last second, respectively – than even TMR – about 39K errors. EDB-Tree CS performs the same in all 5 scenarios but unfortunately has many undetected errors in our simulation, which seems to be some implementation bug we could not eliminate until now. Anyways, it is very sensitive to bit flips as it validates checksums for so many node members and errors in node members which would not be accessed otherwise greatly increase the overall error rate. On the one hand, for 2 and 4 bit flips EDB-Tree PB detects no errors since they cancel each other out as mentioned before. On the other hand, for 1, 3, and 5 bit flips it detects even more bit flips than TMR with no memory overhead and very little performance penalty, as shown in the experiment before. Nevertheless, the TMR B-Tree variant is able to return no erroneous results in this setup, i.e. there were no undetected errors for all scenarios. As EDB-Tree only

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

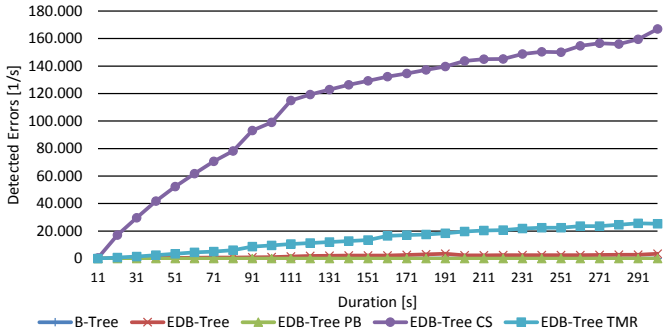

Figure 11: Detected errors for 4 bit flips per 8-Byte word.

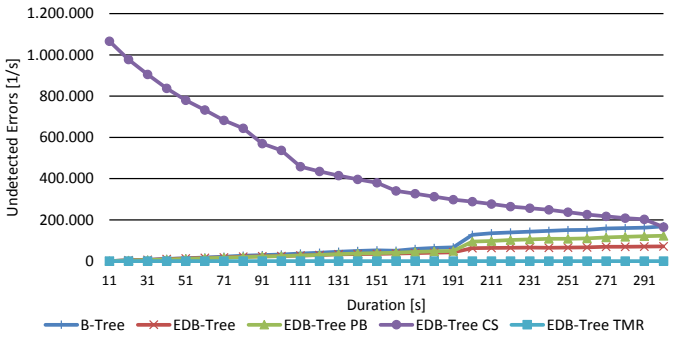

Figure 12: Undetected errors for 4 bit flips per 8-Byte word.

checks pointers, it detects very few errors and returns a relatively high number of wrong values and false negatives, i.e. that the key was not found.

6. CONCLUSION

Main memory errors in the form of single and multi-bit flips are already occurring in current main memory for diverse reasons. With future hardware this trend will only increase. Current solutions like ECC will not scale for different reasons – they are static, general purpose mechanisms which are not able to leverage application knowledge. We presented software based adaptations for B-Trees, a widely used database index structure, to cope with increasing bit flip rates in main memory. We showed that pointer san-

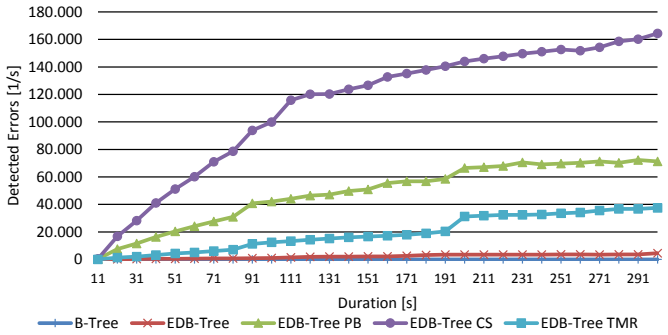

Figure 13: Detected errors for 5 bit flips per 8-Byte word.

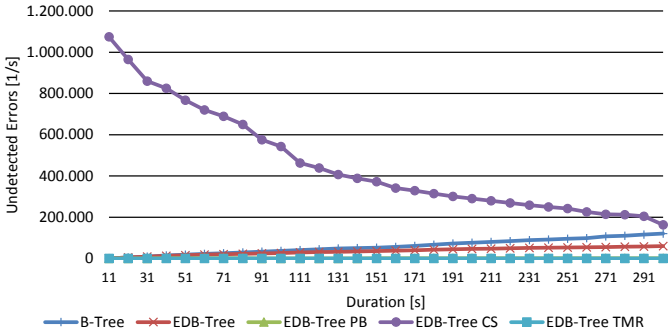

Figure 14: Undetected errors for 5 bit flips per 8-Byte word.

ity checks, parity bits and checksums can deliver comparable or better error detection on commodity hardware compared to ECC hardware, since they are able to detect more than 2 bitflips in 8-byte words. Furthermore, we showed that checksums are able to detect more bit flips and provide higher reliability which is highly desired for database systems. Aside from error detection, the area of error correction is highly relevant, whereas correction approaches using context knowledge of databases are required.

This paper is only a first step towards B-Trees which are resilient against arbitrary main memory bit flips. In future work we want to employ online error correction mechanisms and optimizations like multi-threading awareness. Also, further error detection mechanisms like a combination of parity bits and checksums could be possible.

Acknowledgments

This work has been supported by the state of Saxony under grant of ESF 100111037 (SREX) and the German Research Foundation within cfAED cluster of excellence and the grant LE 1416/22-1 (HEAC).

References

[1] Y. Aumann and M. A. Bender. Fault tolerant data structures. FOCS '96.

[2] M. Böhm, W. Lehner, and C. Fetzer. Resiliency-Aware Data Management. *PVLDB*, 4(12), 2011.

[3] L. Borucki, G. Schindlbeck, and C. Slayman. Comparison of accelerated DRAM soft error rates measured at component and system level. In *IRPS 2008*.

[4] G. S. Brodal et al. Optimal Resilient Dynamic Dictionaries. In L. Arge, M. Hoffmann, and E. Welzl, editors, *Algorithms âĂŞ ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 347–358. Springer Berlin Heidelberg, 2007.

[5] H. T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition edition, 2009.

[6] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient Search Trees. SODA '07.

[7] I. Finocchi and G. F. Italiano. Sorting and Searching in the Presence of Memory Faults (without Redundancy). STOC '04.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art Nr. 5, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619233

[8] E. Fujiwara. *Code Design for Dependable Systems: Theory and Practical Applications*. Wiley Interscience, 2006.

[9] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *Knowledge and Data Engineering*, 4(6), 1992.

[10] G. Graefe and R. Stonecipher. Efficient verification of b-tree integrity. In *BTW*, pages 27–46, 2009.

[11] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic Rays DonŠt Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. *SIGARCH Comput. Archit. News*, 40(1), 2012.

[12] A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority Queues Resilient to Memory Faults. In *WADS 2007*.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[14] K. Küspert. Efficient Online Error Detection Techniques for Trees in Database Systems. In *Fehlertolerierende Rechensysteme*, volume 84 of *Informatik-Fachberichte*. 1984.

[15] J. Lin, H. Zheng, Z. Zhu, H. David, and Z. Zhang. Thermal Modeling and Management of DRAM Memory Systems. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[16] T. K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. 2005.

[17] E. Normand. Single Event Upset at Ground Level. *IEEE transactions on Nuclear Science*, 43(6), 1996.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[19] B. Schroeder and G. A. Gibson. A Large-scale Study of Failures in Highperformance-computing Systems. *Dependable and Secure Computing*, 7(4), 2010.

[20] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull*, 36(2), 2013.