TLB misses — the Missing Issue of Adaptive Radix Tree?

Petrie Wong[§] Ziqiang Feng[†] Wenjian Xu[†] Eric Lo[†] Ben Kao[§]
[§] Department of Computer Science, The University of Hong Kong
[†] Department of Computing, The Hong Kong Polytechnic University
[§]{kfwong2, kao}@cs.hku.hk
[†]{cszqfeng, cswxu, ericlo}@comp.polyu.edu.hk

ABSTRACT

Efficient main-memory index structures are crucial to main-memory database systems. Adaptive Radix Tree (ART) is the most recent in-memory index structure. ART is designed to avoid cache miss, leverage SIMD data parallelism, minimize branch mis-prediction, and have small memory footprint. When an in-memory index structure like ART has significantly few cache misses and branch mis-predictions, it is natural to question whether misses in Translation Lookaside Buffer (TLB) matters. In this paper, we try to confirm whether this is the case and if the answer is positive, what are the measures that we can take to alleviate that and how effective they are.

1. INTRODUCTION

Efficient main-memory index structures are crucial to main-memory database systems such as H-Store [7] and Hekaton [3]. Examples of fast in-memory index include Cache-Sensivitve B$^+$-Tree (CSB$^+$-Tree) [15], Fast Architecture Sensitive Tree (FAST) [9], and Adaptive Radix Tree (ART) [11]. CSB$^+$-Trees ensure their nodes fit into multi-level of cache so as to reduce the cache miss rate. In addition to reducing cache miss, FAST leverages Single Instruction Multiple Data (SIMD) instructions to carry out data operations in parallel. ART is the most recent in-memory index structure. Similar to CSB$^+$-Tree and FAST, ART is also designed to avoid cache miss and leverage SIMD data parallelism. Furthermore, ART limits its tree height by adopting a radix tree structure, thereby largely reducing its branch mis-prediction cost. Overall, experiments have shown that ART outperforms all existing in-memory index structures in both search and update with a small memory footprint [11]. Currently, ART is the key index structure in HyPer [8].

When an in-memory index structure like ART has significantly fewer cache misses and branch mis-predictions, it is natural to question whether misses in Translation Lookaside Buffer (TLB) would become a bottleneck [1]. In this paper, we try to confirm whether this is really the case and if the answer is positive, what are the measures that we can take to alleviate that and how effective those measures are.

DaMoN'15, May 31 - June 4, 2015, Melbourne, VIC, Australia

http://dx.doi.org/10.1145/2771937.2771942 .

2. BACKGROUND OF ART

Adaptive radix tree (ART) [11] is a high performance and space-efficient general purpose index structure for main memory databases, specially tuned for modern hardware.

An example of an ART index is shown in Figure 1. It is a byte-wise radix tree that uses the individual bytes of a key for indexing. As a result, all operations have a complexity of $O(k)$, where k is the key length in byte. The ART in Figure 1 is used to index 3-byte keys. Keys in ART are stored implicitly and are represented by the paths from the root to the leaves. The i-th level indexes the i-th byte of the keys. For example, to lookup a value indexed by key `0x010203` in Figure 1, it first looks up the first byte `0x01` of the key in the first level (the root), follows the pointer to the second level and looks up the second byte `0x02`, and then follows the pointer to the third level (the leaf) and looks up the last byte `0x03`. As ART implicitly stores keys in lexicographical order, it supports not only exact lookups but also range scans and prefix lookups. In order to reduce the memory footprint, ART collectively supports nodes of four different sizes. The nodes that index the key `0x010203` are of type Node256, which is simply an array of 256 pointers. With that representation, the next node can be efficiently found using a single lookup of the next key byte. However, if many entries are null, that representation is not space-efficient. Another type of node is called Node4, which consists of an array of length 4 for keys and another array of the same length for pointers. In Figure 1, when looking up a value indexed by key `0xFDEEFF` it first looks up the byte `0xFD` of the key in the first level (which is of type Node256) and follows the pointer to the second level (which is of type Node4). Within that node, the second byte `0xEE` of the key is first searched within the key array using SIMD 4-way instructions. Then it follows the pointer of the corresponding entry in the pointer array to the third level (the leaf). In Figure 1, the one that holds the last byte `0xFF` of the key `0xFDEEFF` is of type Node48. Node48 does not store the keys explicitly. Instead, a 256-element array is used, which can be indexed with key bytes directly. The element in the array points into a second array which contains up to 48 pointers. Node48 is used to hold between 17 and 48 child pointers. This indirection saves space in comparison to Node256 because each element requires only 6 bits ($2^6 \geq 48$). The last type of node which does not appear in Figure 1 is Node16. This type of node is similar to Node4 except that its arrays

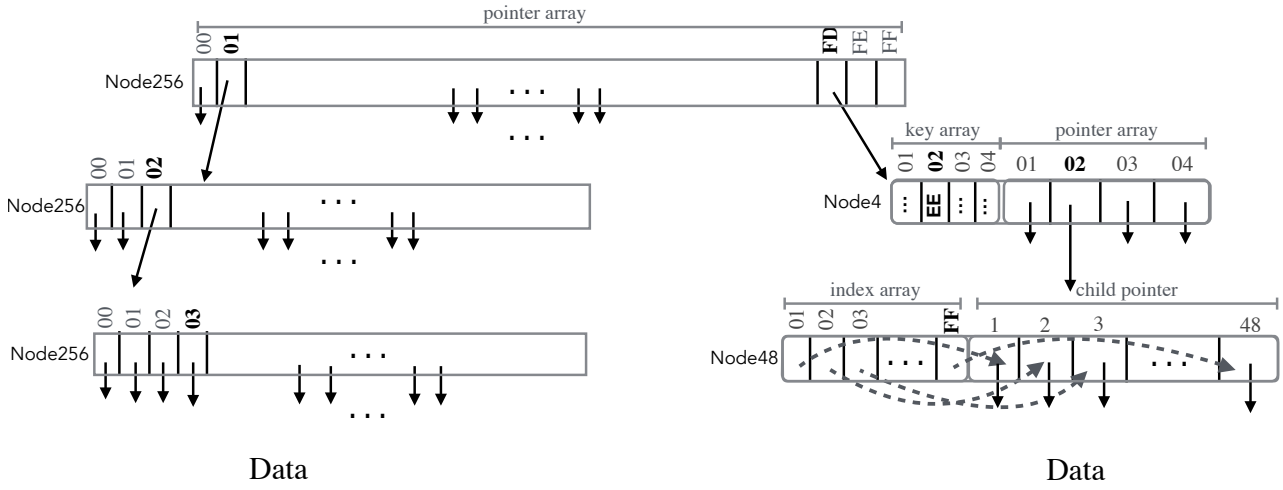


Figure 1: Adaptive Radix Tree

have space for 16 entries.

ART chooses the node type depending on the number of non-null children. For example, when a node of type Node4 is full, ART will replace that with a node of type Node16. Generally, ART has few branch mis-predictions because byte matching within Node48 and Node256 is done through offset calculation instead of comparison. Furthermore, its memory footprint is smaller when comparing with a pure radix tree because of the use of adaptive node size, which also leads to fewer cache misses.

3. TLB MISSES IN ART

Figure 2 shows the average stall time due to TLB miss as a percentage of the latency of a lookup in ART. We ran the experiments on a desktop with an Intel Core i7 2630QM CPU, which has 4 cores, 8 threads, 2.00 GHz clock rate, and 2.9 GHz turbo frequency. Each core has 256KB unified L2 Cache, 32KB L1i cache and 32KB L1d cache. All cores share 6MB L3 cache and 16GB 1600 RAM. We used Linux 3.2 in 32-bit mode as operating system and GCC 4.6 as complier.

We generated 32-bit integers as the keys. By default, we randomly generated $n = 1,000,000$ keys. Following [11], we generated both *dense* keys and *sparse* keys. Dense keys range from 0 to $n-1$ and sparse keys range from 0 to $2^{32}-1$. The sizes of ART under dense keys and sparse keys are 19 MB and 22 MB, respectively.

For real OLTP workloads, key accesses often posses certain skewness [12, 17]. So we generate index lookup workloads using Zipfian distribution, where a zipf factor $zipf = 0$ means a uniform distribution (i.e., all keys are equally hot) and a very high $zipf = 3$ factor means there are only a few items that are very hot. Each workload consists of 256 million ART index lookups.

From Figure 2 we see that the stall time due to TLB miss constitutes up to 23% of the overall index lookup time. When the workload is unreasonably skew ($zipf = 3$), there are only one or few very hot items and very few page table entries (PTE) are needed. Consequently, TLB could afford to keep all those entries and almost no TLB miss is incurred. However, when the skewness is close to what realistic workloads should possess (e.g., $zipf = 1$ to 2), we see the stall time due to TLB miss would become a non-negligible factor of the whole index access latency. Having seen that, we next ask if we can alleviate that issue effectively.

4. CAN HUGE PAGE HELP?

Our first direction is to investigate whether the use of *huge page* in ART can help. The regular page size of most modern processor architectures is 4KB. However, recent modern processor architectures can support multiple page sizes [6]. For example, Xeon E5 4650 processor can support page sizes of 2MB and 1GB , in addition to 4KB regular page sizes. Previous study has indicated that using huge page is a good tactic to reduce TLB misses (e.g., [2]). So, if ART nodes are stored using huge pages, the number of pages spanned by ART nodes shall be reduced, and that can reduce the pressure on the TLB. However, we would like to point out that modern processors have different number of TLB entries for different page sizes. Table 1 shows the number of TLB entries for Intel processor under Sandy Bridge micro architecture. Processors under that micro architecture has 64 L1 DTLB entries and 512 L2 STLB entries for regular pages (4KB), but it has only 32 L1 DTLB entries for huge pages (2MB). So while the number of pages spanned by ART nodes could be reduced by using huge pages, the number of TLB misses may not be reduced accordingly — after all, the number of huge page entries are fewer than that of regular page entries in TLB. On the other hand, a page table entry would also write-through the processors' L1/L2/L3 cache when a TLB miss occurs. So, when the number of page table entries is reduced, the pressure on the processors' cache is also reduced, leading to fewer cache misses and thus possibly higher throughput. In a nutshell, we can at least come up with three factors that are associated with the use of huge page:

- Factor 1: The use of huge page means fewer page table entries are needed.
- Factor 2: But TLB has fewer slots to hold huge page entries.
- Factor 3: The use of huge page indirectly reduces the pressure on the processors' cache and could possibly lead to fewer cache misses.

TLB		Page Size		
Name	Level	4KB	2MB	1GB
DTLB	1st	64	32	4
ITLB	1st	128	8	none
STLB	2nd	512	none	none

Table 1: TLB Capacity in Intel Sandy Bridge processors

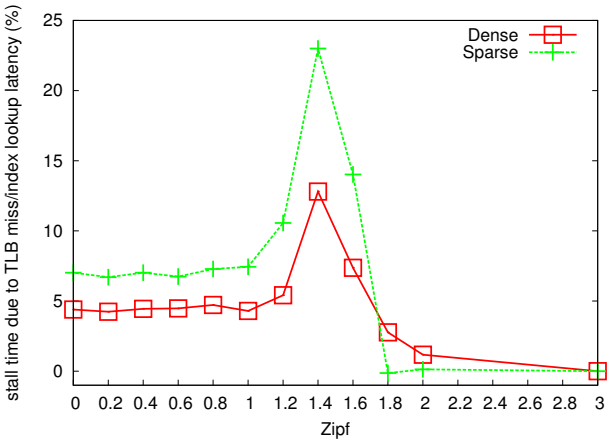


Figure 2: $\frac{\text{Stall time due to TLB miss}}{\text{Index lookup latency}}$ (%)

Factor 1 and Factor 2 are subtle factors of TLB miss. In order to study the influence of the use of huge page to the performance of ART, we carried out experiments to study the improvement of lookup throughput of ART by using (2MB) huge page over regular page. We did not carry out experiments of using huge page of size 1GB because our experimental platform does not support that. Figure 3 illustrates that **the use of huge page provides positive lookup throughput improvement over the use of regular page**.

To understand which factor(s) we mentioned above contribute to the lookup throughput improvement, we present the performance counters of two selected cases: $zipf = 1$ and $zipf = 1.4$ in Table 2. Both cases fall into skewness that realistic workloads possess. From the performance counters, we see that by the use of huge page, there are no more TLB misses (see underlined numbers for a highlight), meaning Factor 1 somehow outweighs Factor 2 in this set of experiments. Furthermore, we see that the use of huge page also reduces the cache misses (Factor 3).

We are particularly interested in knowing why the lookup throughput improvement is especially pronounced (38% improvement for sparse and 21% for dense; the maximum) at $zipf = 1.4$. By looking at Table 2, we see that when $zipf = 1$, L1/L2/L3 cache misses still dominate the number of TLB misses (see the **bold** numbers for a highlight). That is because the number of hot items is still quite a lot and the total size of the hot items is larger than the cache. But when $zipf = 1.4$, the number of L2/L3 cache misses is substantially reduced because the number of hot items is much smaller and they fit into L2/L3 cache. As a result, TLB misses become one of the dominating factors under $zipf = 1.4$. So, when the number of TLB misses are reduced by the use of huge page, that translates to the removal of a dominating factor under $zipf = 1.4$ but the removal of a non-dominating factor under $zip = 1$. That explains why the lookup throughput improvement is especially pronounced under $zipf = 1.4$.

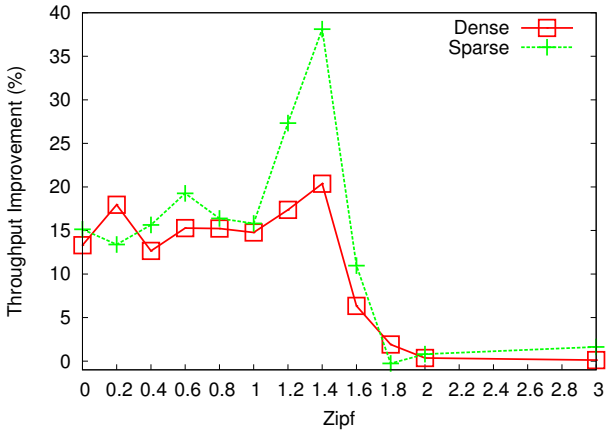


Figure 3: Throughput Improvement by using Huge Page over Regular Page

5. CAN WORKLOAD-CONSCIOUS NODE-TO-PAGE REORGANIZATION HELP?

Normally, tree nodes in ART are allocated through dynamic memory allocation functions such as `malloc`. So, whether two nodes of ART are within the same page is all up to the operating system dynamic memory allocator (e.g., the slab allocator in Linux [16]), whose goal is to eliminate fragmentation over the whole memory space. Recall that in OLTP workloads, key accesses tend to be skewed: some keys are "hot" and accessed frequently, others are "cold" and accessed infrequently. Imagine that if ART is able to put all hot nodes into only one (huge) page, then the page table entry of that huge page shall consistently stay in TLB, meaning there would be almost no TLB miss for all hot key accesses.

In the following, we consider an approach that takes over OS's default control and organize the hot ART nodes into the same page. This approach follows the workload-conscious node reorganization method proposed in [17]: (i) key accesses are logged as part of query execution; (ii) then, the access logs are analyzed (in another thread) to compute the access frequencies of the keys; (iii) finally, a node-to-page reorganization takes place to arrange all keys into the page according to their access frequencies.

In order to reduce the overhead, each work thread samples probabilistically key accesses and writes to a dedicated circular buffer log. Periodically, the log is analyzed to compute the access frequencies of the keys by a separate thread. A node-to-page reorganization process starts after each analysis by creating a new memory space. Then, it starts with the hottest key and performs a lookup of that key on ART. For each node accessed, that node is copied to a page of the new memory space. The process repeats with each key according to their access frequencies, in descending order. When a page of the new memory space is full, it starts another new page in that space.

The advantage of this approach is that it requires no mod-

$zipf = 1$	Dense		Sparse	
	Regular	Huge	Regular	Huge
Cycles	399.30	347.71	418.84	358.65
# Instructions	250.54	249.95	217.04	216.67
TLB visit	59.40	59.59	53.74	53.82
TLB miss	1.32	0.00	**1.89**	0.00
L1 miss	5.63	4.70	**7.37**	5.67
L2 miss	4.21	3.87	**5.21**	4.77
L3 miss	2.27	2.24	**2.84**	2.78
Misp. Branch†	1.82	1.97	1.04	1.49

$zipf = 1.4$	Dense		Sparse	
	Regular	Huge	Regular	Huge
Cycles	125.70	104.18	138.63	100.09
# Instructions	249.98	250.03	216.48	216.59
TLB visit	59.42	59.52	53.82	53.65
TLB miss	0.92	0.00	**1.47**	0.00
L1 miss	5.05	4.27	**6.57**	5.05
L2 miss	0.97	0.50	**1.76**	0.86
L3 miss	0.00	0.00	**0.00**	0.00
Misp. Branch†	1.49	1.54	1.08	1.03

† per 1K lookup

Table 2: Performance Counters Per Lookup

# of keys	512k	1M	2M	4M	8M	16M
Sparse	14	22	60	127	188	310
Dense	10	19	38	77	154	308

Table 4: Memory Usage (MiB) with Varying Data Size

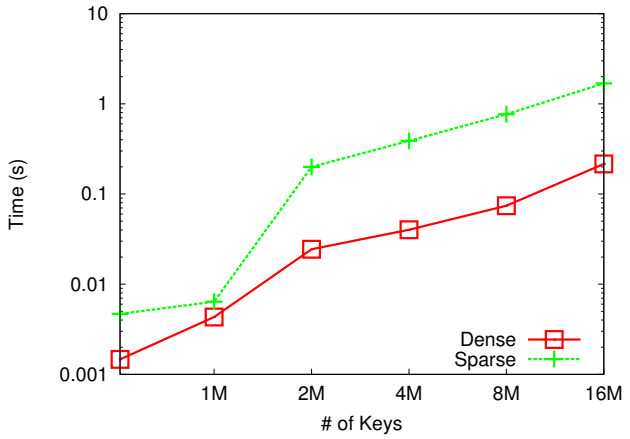


Figure 5: ART Reorganization Time versus Number of Keys

ification to the code of the original ART methods such as `insert()` and `lookup()`. We implement the node-to-page reorganization as an external function `reorganize()`, which is invoked periodically. The downside of this approach is that the tree cannot be updated during node-to-page reorganization takes place.

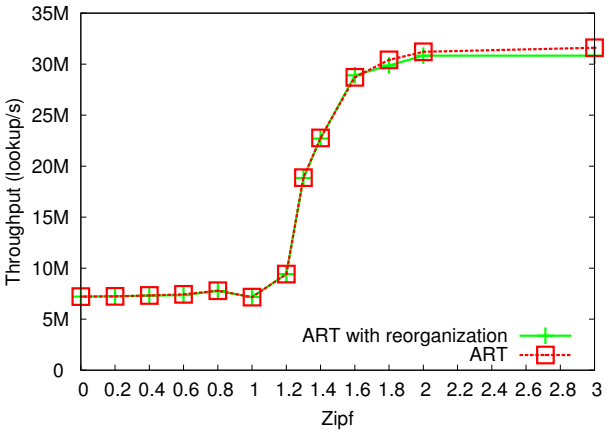
Figure 4 shows the effectiveness of such approach. We report the throughput of ART with or without workload-conscious node reorganization under different workload skewness. The throughput of ART with workload-conscious node reorganization is reported as the throughput after executing 1% of the index lookup requests and carrying out node-to-page reorganization.

From the figure, we observe that the **workload-conscious node reorganization is effective when the data is sparse but is ineffective when the data is dense**. To explain, we have to understand that when the data is sparse, each radix (each ART node) should not contain many children and thus small nodes like Node4 are used. A Node4 consists of 4 bytes offset, 4 pointers, and 16 bytes header, which is about 36 bytes in size. When regular page is used (Figures 4a, c and e), each 4K page can hold about 113 ART nodes. Since there are 512 STLB entries, that means the STLB can buffer $512 \times 113 = 57856$ nodes. In this case, by using workload-conscious node reorganization, those 57856 nodes are surely the most frequent ones and they all get hit in TLB. In contrast, when the data is dense, each radix (each ART node) should full of children and thus Node256 are all used in ART. A Node256 consists of 256 pointers. So, including the header, such a node is least 1KB in size. When regular page is used, each 4K page can hold only 3 ART nodes. Since there are 512 STLB entries, that means the STLB can buffer $512 \times 3 = 1536$ nodes. In this case, by using workload-conscious node reorganization, those 1536 nodes are surely the most frequently ones. Comparing this number (1536) with the one (57856) we got on sparse data, we understand why the benefit of workload-conscious node reorganization under dense data is not as significant as under sparse data, in the context of regular page.

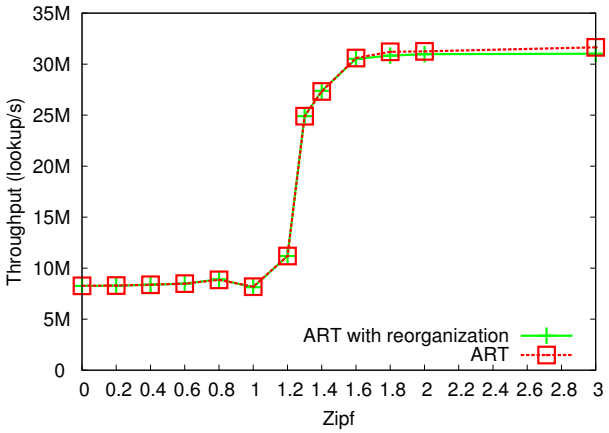
When the data is dense, ART contains fewer but larger nodes like Node256. When huge page is used (Figures 4b, d and f), there would be few pages needed. So all page table entries can stay in TLB, giving almost no TLB miss (refer to Table 3). This makes node-to-page reorganization immaterial (Figure 4b). When the data is sparse, ART contain many small nodes like Node4. So, a huge page can hold many ART nodes but also more pages are needed because of the small nodes usually contain space when no children. Therefore, we observe benefits brought by workload-conscious node reorganization (Figure 4d). We repeat the experiments on 64-bit length keys and they are similar to the 32-bit length keys data set. The results are presented in Figure 4e and f.

Lastly we study how the ART's reorganization cost varies with the number of keys. Figure 5 shows that the reorganization time increases with the number of keys in general. Also, reorganization under sparse keys consumes more time than dense keys. The reorganization time essentially relates to the memory footprint of the ART. Larger memory footprint implies more node movement (reorganization) to be carried out, resulting in longer execution time. As shown in Table 4, the memory footprint increases with the data size in general, because large-size node types need to be used. Moreover, the memory usage under sparse keys grows more rapidly than that of dense keys, which explains the difference between sparse keys and dense keys in Figure 5.

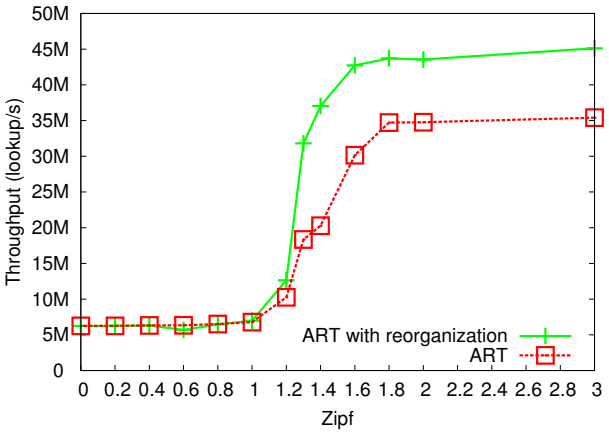
6. RELATED WORK



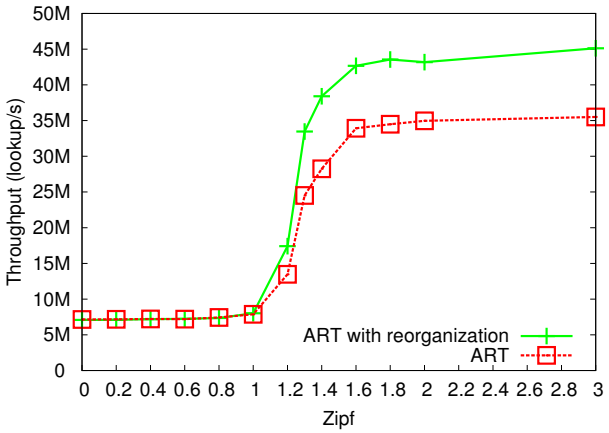
a)Dense / Regular Page



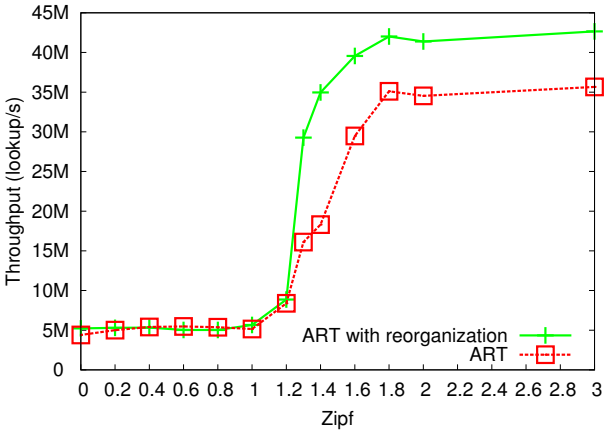
b)Dense / Huge Page



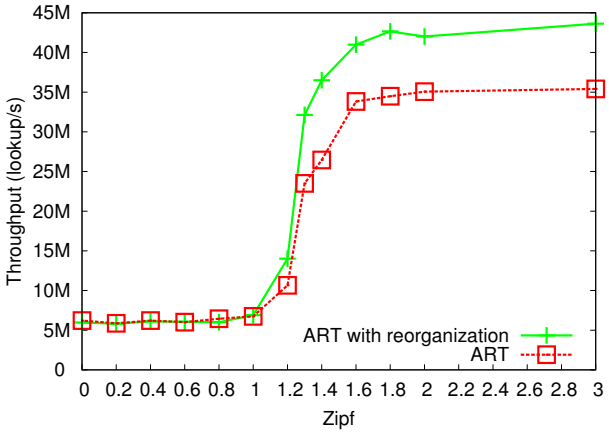
c)Sparse / Regular Page



d)Sparse / Huge Page



e)Sparse (64bit) / Regular Page



f)Sparse (64bit) / Huge Page

Figure 4: Effectiveness of Workload-Conscious Node-to-Page Reorganization

	ART				ART with Reorganization			
	Dense		Sparse		Dense		Sparse	
$zipf = 1$	Regular	Huge	Regular	Huge	Regular	Huge	Regular	Huge
Cycles	392.33	341.96	414.19	356.61	396.44	342.73	356.08	350.06
# Instructions	250.24	249.85	217.24	217.19	250.19	250.00	216.49	216.30
TLB visit	59.50	59.59	53.71	53.67	59.58	59.58	53.37	53.31
TLB miss	1.32	0.00	1.89	0.00	1.34	0.00	1.78	0.00
L1 miss	5.63	4.70	7.32	5.63	5.68	4.73	7.26	5.66
L2 miss	4.21	3.86	5.26	4.75	4.20	3.87	5.15	4.77
L3 miss	2.10	2.08	2.61	2.56	2.10	2.09	2.47	2.44
Misp. Branch†	1.07	1.46	1.53	1.13	2.14	1.61	22.27	21.54
	ART				ART with Reorganization			
	Dense		Sparse		Dense		Sparse	
$zipf = 1.4$	Regular	Huge	Regular	Huge	Regular	Huge	Regular	Huge
Cycles	124.89	103.80	136.02	103.59	99.69	126.97	76.35	74.09
# Instructions	250.29	249.62	216.79	250.12	217.19	249.86	175.07	174.95
TLB visit	59.46	59.47	53.74	53.60	59.45	59.41	37.70	37.61
TLB miss	0.92	0.00	1.48	0.00	0.91	0.00	0.00	0.00
L1 miss	5.05	4.26	6.57	5.04	5.04	4.26	3.25	3.31
L2 miss	0.97	0.49	1.70	0.83	0.95	0.49	0.26	0.01
L3 miss	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Misp. Branch†	1.40	1.42	0.40	0.73	2.23	2.62	0.63	0.74

† per 1K lookup

Table 3: Performance Counters Per Lookup (with/without node-to-page-reorganization)

TLB On modern processors, a virtual address (VA) to physical address (PA) translation must be carried out for each instruction or data access. Such translations are originally serviced by a page table in memory. To accelerate this process, the Translation Lookaside Buffer (TLB) is added to cache a limited number (e.g., 64) of translation entries for the processor's quick access. As a result, each data access request must consult TLB first before the data can be addressed physically. This fact puts the TLB in the critical path of the programs. A TLB miss may then stall the processor and cause wasted resources.

The TLB issue in main memory joins and sorts is first noted in [13] and studied for more recent hardware in [10, 2, 14]. Kim et al. [10] suggested limiting the partition fanout to at most $2 \times N_{\text{TLB}}$ to avoid excessive TLB thrashing. Balkesen et al. [2] demonstrated through experiments that by configuring a large page size one can significantly reduce the number of TLB misses during hash joins. They also advocate using an in-cache buffer to amortize the TLB miss penalty across several items, which is followed by [14].

Zhou and Ross [18] and Hankins and Patel [5] identified the important impact of TLB performance on memory-resident index trees. Hankins and Patel [5] suggested there is a balance to strike between reducing cache misses and reducing TLB misses in CSB$^+$-tree [15]. The Fast Architecture Sensitive Tree (FAST) is (re)designed carefully to reduce both cache misses and TLB misses [9].

Workload-conscious Making data structures workload-conscious is another way to reduce hardware penalty. For example, data morphing [4] arranged the data layout based on analysis of the query workload, so that a workload of heterogeneous operations can be executed in a cache-efficient manner. Stoica and Ailamaki [17] used access logs to identify hot data and cold data in the database. They then reorganized the data items in memory pages to separate hot and cold data into different regions. As a result, disk I/O's are reduced and hit rate is improved.

7. CONCLUSIONS

In this paper, we study whether TLB miss would be an important factor under ART, given that ART is good at reducing cache misses and branch mis-predictions. Our experiments show that **TLB miss does matter when the access workload possess realistic skew**. Correspondingly, we first study whether the folklore of using huge page help. Our experiments illustrates that **the use of huge page provides positive lookup throughput improvement over the use of regular page**. We also study whether we can carry out workload-conscious node-to-page reorganization to reduce TLB miss. Our experiments illustrates that **that helps when the data to be indexed is sparse**.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, 2013.

[3] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[4] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03. VLDB Endowment, 2003.

[5] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. *SIGMETRICS Perform. Eval. Rev.*, 31(1), June 2003.

[6] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, 2014.

[7] 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.

[8] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[9] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[10] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. 2009.

[11] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[12] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 26–37. IEEE, 2013.

[13] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? dissecting cpu and memory optimization effects. In *Proceedings of the 26th international conference on very large data bases*, pages 339–350. Morgan Kaufmann Publishers Inc., 2000.

[14] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 755–766, 2014.

[15] J. Rao and K. A. Ross. Making b+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000.

[16] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2005.

[17] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, 2013.

[18] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 405–416. VLDB Endowment, 2003.