OLTP in Wonderland

Where do cache misses come from in major OLTP components?

Pınar Tözün
EPFL
pinar.tozun@epfl.ch

Brian Gold
Oracle Labs
brian.gold@oracle.com

Anastasia Ailamaki
EPFL
natassa@epfl.ch

ABSTRACT

For several decades, online transaction processing has been one of the main applications that drives innovations in the data management ecosystem, and in turn the database and computer architecture communities. Despite the novel approaches from industry and various research proposals from academia, recent studies emphasize that OLTP workloads still cannot exploit the full capability of modern processors.

To better integrate OLTP and hardware in future systems, we perform a detailed analysis of instruction and data misses, the main causes of memory stalls. We demonstrate which operations and components of a typical storage manager cause the majority of different types of misses in each level of the memory hierarchy on a configuration that closely represents modern commodity hardware. We also observe the impact of data working set size on these misses.

According to our experimental results, **L1 instruction** misses are an extensive cause of the overall stall time for OLTP even for data working set sizes as large as 100GB as long as the data fits in memory. **Capacity** misses coming from the **index probe** operation are the dominant cause of the instruction and data stalls when running typical OLTP workloads. During index probe (one of the most common operations in OLTP), the **B-tree**, **lock**, and **buffer** management components of a storage manager are responsible for more than half of the total misses.

1. INTRODUCTION

Online transaction processing (OLTP) is one of the most important data management applications. As a result, it leads fundamental new research and system development efforts in both computer architecture and data management communities [11, 19].

Despite recent advances in transaction processing and computer architecture, previous studies [2, 6, 16, 24, 26, 28] analyzing the micro-architectural behavior of OLTP workloads on modern hardware emphasize that OLTP exploits modern micro-architectural resources very poorly. Most of the execution time (∼80%) goes to memory stalls [6]; as a result, on processors that have the ability to execute four instructions in a cycle, which is the most common on modern commodity hardware, OLTP achieves around one instruction per cycle (IPC) [28]. Such under-utilization of micro-architectural features is a great waste of hardware resources.

Several proposals have been made to reduce memory stalls through increasing cache hit rates. These range from cache-conscious data structures and algorithms [4, 8] to sophisticated data partitioning and thread scheduling [22] for data, and from compilation optimizations [23], advanced prefetching [7], to computation spreading [1, 3] for instructions. Although these techniques reduce data or instruction misses to a great extent, some aiming specifically for OLTP workloads and some for more general applications, none of them has detailed insights on why misses in OLTP happen and where they come from within the storage manager.

In this work, we thoroughly analyze the data and instruction misses of an OLTP system to answer the following questions: (1) What types of database operations; *scan*, *index probe* etc., and which parts of a storage manager; *locking*, *logging* etc., are responsible for various kinds of misses? (2) How sensitive are the results to the data working set size of the workloads? Our aim is to give insights and hints to researchers and developers trying to optimize their code, data, or utilization of the hardware resources for minimizing memory stalls while running OLTP.

Using Pin [17], we extract instruction, data, and function traces from the Shore-MT storage manager [14] while running the OLTP benchmarks standardized by the Transaction Processing Performance Council (TPC) [29]. We replay the traces on a cache configuration that is typical for modern commodity hardware and give miss rates, types, and breakdowns for the main storage manager components. Our contributions are listed below:

- We show that the **L1 instruction cache** misses account for a significant part (40-80%) of the overall stall time even when the memory-resident data working set size increases (from 0.1GB to 100GB).
- We demonstrate that the **capacity** misses are the single dominant factor in stalls since the cache associativities of typical modern hardware is sufficient to minimize the conflict misses for both data and instructions.
- We identify the **index probe** operation as the leading component of the cache misses. We also highlight the **B-tree**, **lock**, and **buffer** managers as the storage manager

DaMoN'13, June 24 2013, New York, NY, USA

parts that contribute to most of the instruction (∼55%) and data (∼60%) misses during an index probe.

The rest of the paper is organized as follows: Section 2 surveys related work in more detail. Section 3 describes our experimental methodology. Section 4 presents a sensitivity analysis on the data size. Section 5, first classifies the most problematic misses into *conflict*, *capacity*, and *compulsory* ones, and then, associates various instruction and data misses into storage manager operations and components. Finally, Section 6 concludes the paper by summarizing the results and discussing possible solutions to minimize stalls.

2. RELATED WORK

There is a large body of related work that analyzes various OLTP workloads from low-level hardware-side analysis, e.g. workload characterization studies, to high-level software-side ones, e.g. time breakdowns.

Previous work on workload characterization investigates OLTP workloads at the micro-architectural level. Barrosso et al. [2] study the memory system behavior of OLTP. Ranganathan et al. [24] and Keeton et. al. [16] focus on the effectiveness of out-of-order execution on SMPs for OLTP workloads by using a simulation environment and a real machine, respectively. They all conclude that OLTP cannot exploit aggressive micro-architectural features, wasting most of its time in memory stalls and exhibiting low IPC.

More recent workload characterization studies [6, 28] examine the behavior of OLTP workloads on modern commodity hardware. They show the same high-level conclusions with the older workload characterization studies demonstrating that, after almost 15 years, OLTP still cannot fully exploit the micro-architectural resources of the most commonly used hardware types today, despite the advances in both the database and computer architecture communities. Even though these studies highlight the lower level problems of OLTP on modern hardware, there is no clear attribution of the hardware-side problems to the software-side components of a typical OLTP system.

Johnson et al. [13, 15] and Pandis et al. [20, 21] provide time breakdowns for typical OLTP benchmarks showing where they spend the most of their execution time in the storage manager. Their primary goal is to identify components that are scalability bottlenecks on modern hardware and propose alternative design decisions to remove those bottlenecks. We provide similar breakdowns to spot the storage manager components that are responsible for the majority of data and instruction stalls.

Harizopoulos et. al. [9] detail where the time goes within the storage manager during a single threaded execution in an OLTP system. They demonstrate that logging, latching, locking, and buffer pool altogether take 75% of the total execution time. VoltDB [30], the commercial version of the H-Store system [27], is designed based on these findings. H-Store specifically aims to increase performance by eliminating all four problematic components with an in-memory shared-nothing system design where each partition only has a single worker thread. This paper provides similarly valuable insights that complement this previous work by mapping cache misses to storage manager components, thereby guiding future software and hardware system designs on how to minimize memory stalls.

Table 1: Server Parameters

Processor	Intel Xeon E5-2660
#Sockets	2
#Cores per Socket	8 (OoO)
#HW Contexts	32
Clock Speed	2.2GHz
Memory	125GB 167-cycle access latency (avg. of remote and local)
L3/LLC (shared)	20-way 20MB 19-cycle access latency
L2 (per core)	8-way 256KB 8-cycle access latency
L1 (per core)	8-way 32KB, split I/D 4-cycle access latency
Core width	4-wide retire and issue

3. SETUP AND METHODOLOGY

We perform a trace simulation study rather than working with hardware counters on real hardware. This allows us to (1) change some of the hardware parameters (like in Section 5.1), and (2) have the detailed function call information to map the various cache misses to software components.

Simulator. We build a custom trace simulator to replay the traces and calculate miss rates on various cache configurations. For this study, we model the memory hierarchy of an Intel Xeon E5-2660 server, see Table 1 for details [12].

Traces. The data, instruction, and function name traces are collected from Shore-MT using Pin [17], which can instrument x86 binaries. Pin is only able to instrument application level code; therefore, the Pin traces do not include the system-level instructions. To measure the effect of different storage manager components on cache misses, however, the application level trace contains all the necessary information. Moreover, since the data working set size is memory-resident throughout the experiments, the system time is very low (application time is 200X more than the system time).

Workloads. The traces are collected for three transaction processing benchmarks standardized by TPC [29]; TPC-B, TPC-C, and TPC-E, while running their workload mix on Shore-MT storage manager [14, 25]. Except where indicated in Section 4, we use 100GB databases. The buffer-pool is set big enough to keep the whole database in memory and the log is flushed to RAM due to not having a suitably fast I/O subsystem. Allowing I/O in our analysis would cause an unreasonable bottleneck considering our infrastructure, and therefore, lead us to unrealistic micro-architectural conclusions. To further make sure we run the most optimal configuration possible, all the logging (Aether [15]) and locking (SLI [13]) optimizations of Shore-MT are enabled.

We run a single worker thread while executing transactions. Ideally, scalable multi-threaded execution would avoid most of the data sharing and impose low contention. In turn, the instruction and data streams would not be extremely different between single- and multi-threaded execution. For the instructions, high contention, due to bad initial configuration, would cause threads to spin waiting to acquire locks. This would artifically increase the instruction cache hit rate and give misleading micro-architectural results. For the data, cache coherence related misses would increase under high contention due to extensive data sharing. In this

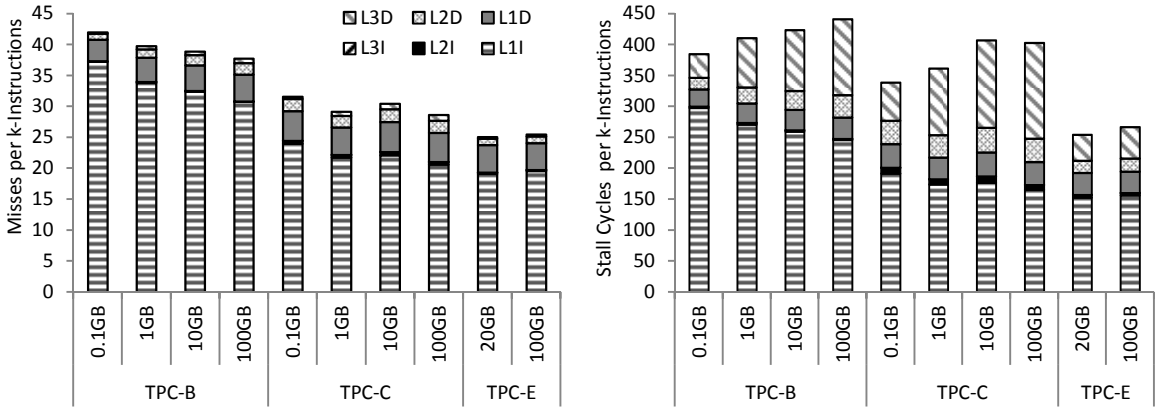

Figure 1: Effect of data size on MPKI (left-hand side) and stall time (right-hand side).

study, we would like to focus on the behavior of an optimally configured storage manager that runs under low contention on the hardware we simulate.

Experiments. We collect two trace files for each workload, where each file contains traces of 1000 different transaction instantiations from the workload's transaction mix. One of the trace files from the same workload is run initially to account for cache warm-up. Then, the simulator starts collecting statistics for cache misses while running the other trace file. All the simulated caches use a LRU replacement policy and 64B cache lines.

To calculate the stall cycles due to cache misses, we multiply the number of misses with the expected penalty for that particular miss as given in Table 1. For LLC misses, we average the penalty for going to local and remote memory.

In stall time breakdowns, we do not account for the possible overlaps of different execution components that would normally happen on a superscalar out-of-order (OoO) processor [5], like the one this paper models. Therefore, even though we draw the stall times on top of each other, some are actually hidden either by other stalls or useful execution. For instructions, the decoupled front-end and back-end of a core would be able to hide some of the stalls. For data, out-of-order execution can hide some of the data stalls while prefetching would reduce the effect of some of the data misses. Nevertheless, although such overlaps can reduce the stall time due to misses, the relative breakdown of the software-side components would be similar even with more complex models that would account for the overlaps. Besides, considering the low IPC of the OLTP workloads [28], we can also assume that not much of the work is overlapped.

4. SENSITIVITY TO DATA SIZE

We initially investigate the effect of increasing data size on the instruction and data misses and stalls coming from different parts of the cache hierarchy. Figure 1 shows the misses per 1000 instructions (MPKI) on the left-hand side and the stall time they cause on the right-hand side for all the workloads. We pick scaling factors that populate around 0.1GB, 1GB, 10GB, and 100GB data for both TPC-B and TPC-C. Since a scaling factor of one already creates ∼20GB data for TPC-E, we run TPC-E with 20GB and 100GB data only.

Looking at the MPKI values in Figure 1, we see that L1 instruction misses dominate the total number of misses regardless of the data size. The domination of the instruction misses also affects the stall time breakdown as shown in Figure 1. Even with 100GB data size, on average 50% of the stalls are because of the L1 instruction misses. On the other hand, L1 and L2 caches, together, are sufficient to keep most of the instruction working set of the workloads we evaluate, keeping the rate of instruction misses from L2 and L3 caches low (at most 2% of the stalls).

Long-latency data misses from L3 caches are the next significant component in the total stall time, even though they form only ∼2% of the total MPKI. As expected, L3 data misses increase as we increase the data size and for 100GB data, around 30% of the stalls are due to L3 data misses. On the other hand, L1 and L2 data misses are not as problematic and probably can be overlapped by out-of-order execution with other outstanding data misses or execution of another instruction.

Compared to the other workloads, TPC-E observes fewer data and instruction misses even though the general trends in different types of misses are very similar for all the workloads. This trend corroborates previous results [6, 28] and can be attributed to increased number of scan operations from simpler workloads, like TPC-B, to more complex ones, like TPC-E. During a file or an index scan, the routine eventually converges to only fetching the next tuple, which has lower instruction footprint than an index probe operation from B-tree root to leaves. Moreover, a file or an index page is scanned from start to end so almost all the parts of the cache lines brought from a database page are touched leading to lower data MPKI.

We also see a decrease in L1 instruction cache MPKI, especially for TPC-B, as we increase the data size. This might stem from the short loop statements in some of the sub-routines of various database operations that needs to iterate more as there is more data. For example, the loop statement in the binary search sub-routine within the index probe operation would have more iterations if there are more data on a particular index page. As a result, the same small instruction working set is executed more frequently at a given time increasing the chances of finding the required instructions in L1-I and reducing the instruction MPKI.

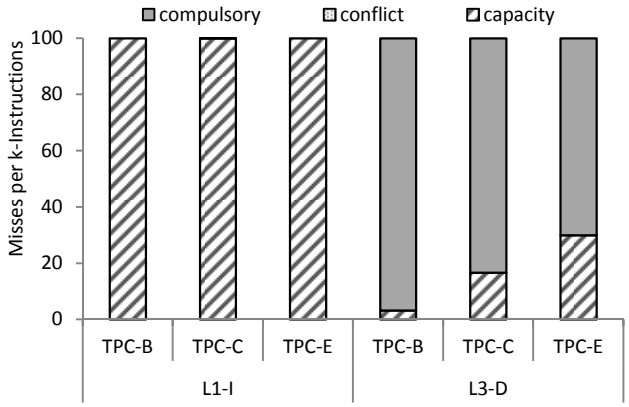

Figure 2: Misses per k-instructions (MPKI) breakdown into compulsory, capacity, and conflict misses.

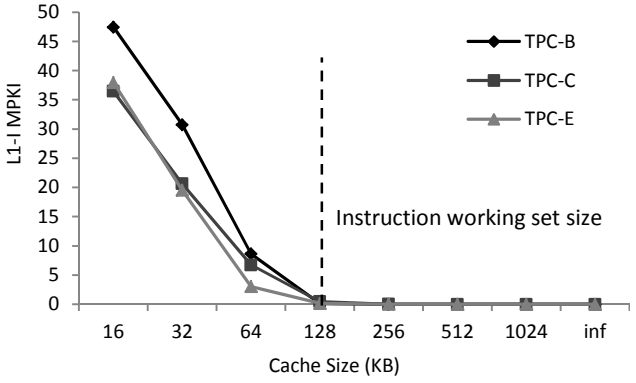

Figure 3: 8-way L1-I MPKI as cache size increases.

5. BREAKDOWN OF MISSES

After examining the total MPKI, this section presents why the most problematic misses happen and where they come from within the storage manager. More specifically, we give breakdowns of the instruction and data misses and stalls for each level of the cache hierarchy in three different granularities: (1) *3C* miss categories (compulsory, capacity, and conflict), (2) database operations (index probe, tuple update etc.), and (3) storage manager components (lock manager, log manager etc.).

5.1 Into Miss Categories

Figure 2 breaks the instruction and data MPKI of the most problematic misses, which are the L1-I and L3 data misses as shown in Section 4, into the *3C*-categorization [10]: (1) *Compulsory* misses are the ones that are missed even with an infinite cache, (2) *Capacity* misses are the extra misses a fully-associative cache observes on top of the compulsory misses, and (3) *Conflict* misses are the ones that happen due to two addresses mapping to the same cache set and replacing one another due to low cache associativity.

As we can see from Figure 2, L1 and L3 cache associativities of the architecture we model, which are 8-way and 20-way, respectively, are sufficient to eliminate all the conflict misses. This leaves the capacity misses as the single cause of all the L1 instruction cache misses on this hardware whereas

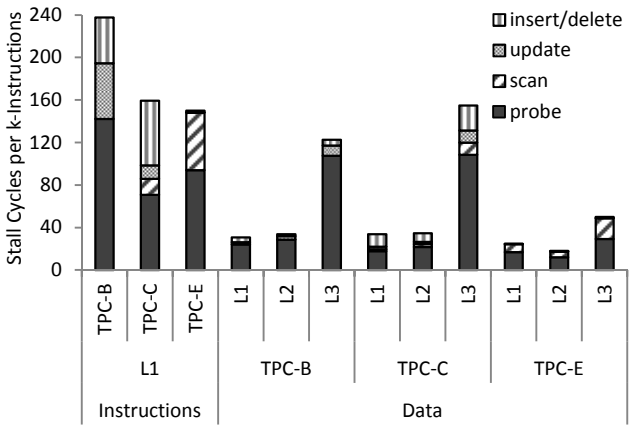

Figure 4: Misses breakdown into database operations at each level of the cache hierarchy.

compulsory data misses dominate the total L3 data misses. After the warm-up run with 1000 transaction traces, the infinite instruction cache basically captures all the instructions needed for these workloads. On the other hand, the data working set size is a lot more than what is accessed in 1000 transactions since the workloads, mostly randomly, access data from a working set of 100GB in our experiments. This explains why there are still many compulsory misses for data even after the warm-up run while we observe none for instructions. If we run longer traces, the percentage of the compulsory misses would be reduced while the capacity misses increase for data as well.

Finally, Figure 3 shows the instruction MPKI for an 8-way L1-I cache as the cache size increases. From Figure 3, one can naïvely think that enlarging the L1-I cache should solve the problem of capacity misses since the instruction footprint of the workloads we evaluate seems to be around 128KB. However, increasing L1-I size also increases the time and energy spent while trying to find an item in the cache. This, in turn, would affect the clock frequency of a processor. Therefore, despite the growing sizes of L2 and L3 caches, today's typical high-performance processors limit their L1 cache sizes to about 32KB.

5.2 Into Operations

In Figure 4, we see the instruction and data stalls in 1000 instructions coming from the three levels of the cache hierarchy separated into different database operations. Since there is either none or very few instruction misses coming from L2 and L3 caches for all the workloads, Figure 4 has breakdowns only for L1 misses for the instructions.

Figure 4 shows that the majority of the misses happen during the index probe operations. This is expected since OLTP workloads do not access many records from a table in their transactions and, hence, they highly depend on the index lookups and scans. The index lookups are especially problematic since the code-path is long and complex. It is interleaved with the function calls to many different modules encapsulating code and data from B-tree, lock, and buffer pool management as Section 5.3 also shows.

While in instruction stalls, we see other major contributors, for data, the index probe seems to be the only significant operation responsible for the stalls. Update, insert,

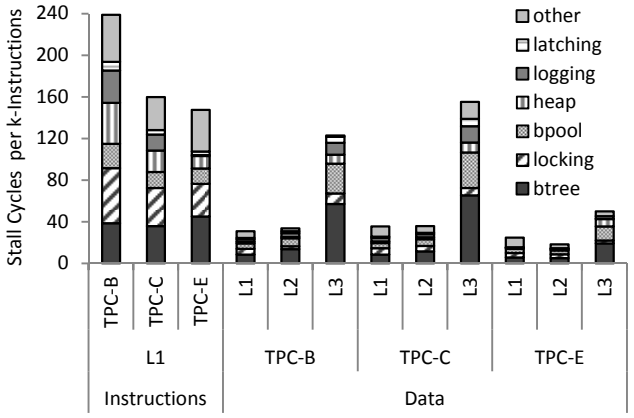

Figure 5: Misses breakdown into storage manager components at each level of the cache hierarchy.

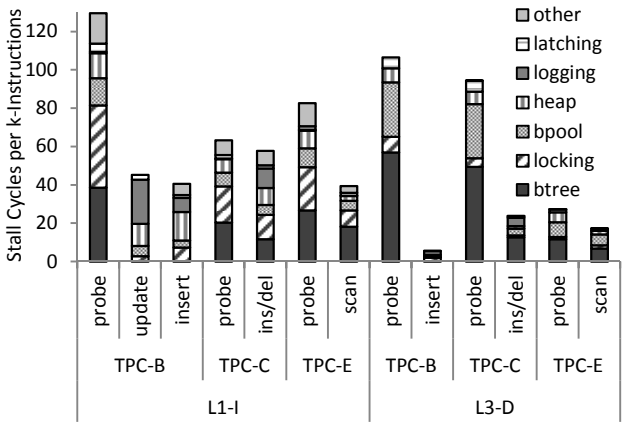

Figure 6: Misses breakdown into storage manager components for each database operation.

and delete operations typically access a single tuple, hence a single heap page, whereas during an index probe several index pages are accessed. In the case of index scans, even though initially there is a probe to find the start point for the scan, afterward the same index and heap pages are re-used frequently increasing the hit rates.

TPC-B is an update-heavy workload and has no index scans. Therefore, updates and inserts are the only operations causing the stalls for TPC-B after the index probes. Going from TPC-B to TPC-E, however, index scans form a bigger portion of the overall stall time as a result of increasing number of scan operations. For TPC-E, we do not see many misses due to read-write operations like tuple inserts, deletes, and updates since majority of the transactions (77%) in its transaction mix are read-only. The trends in the breakdowns, on the other hand, do not change much for different cache levels within each benchmark.

5.3 Into Components

Figure 5 depicts stalls from different types of caches as does Figure 4, but it classifies them into storage manager components rather than database operations. Instruction stalls in L2 and L3 are again omitted since there is either none or very few of them.

Figure 5 does not identify a single dominant component as the cause of instruction stalls. B-tree index operations and lock manager together form ∼45% of the instruction misses on average. Next come the buffer pool and heap manager with ∼23%. For TPC-B, heap manager also takes a significant time due to update and insert heavy nature of this workload. These results corroborate with our findings in Section 5.2, where we show that the index probe operation is the main cause of the instruction and data stalls. The index probe traverses a B-tree from root to leaves and this process is heavily interleaved with the concurrency control mechanism of databases, which is based on ARIES/IM [18] by default in Shore-MT.

For the data stalls, we see the B-tree and buffer pool as the two significant factors, causing more than half of the data stall time for each of the caches. This result also matches with our findings in Section 5.2 since the index probe operation requests many B-tree pages from the buffer pool during the traversal.

We also give a breakdown within the basic database operations to see which storage manager components affect the stall time during these operations. Figure 6 shows this breakdown for the L1 instruction cache and L3 data stalls since, in Section 4, we identify them as the leading causes of the overall stalls. The operations that do not contribute much to the stall time are omitted for simplicity.

Inside the index probe and scan related stall times, we see B-tree, lock manager, and buffer pool as the dominant components for both L1 instruction and L3 data misses. As we have also mentioned above, this result is expected for the index probe operation considering its characteristics. For the update or insert/delete operations, however, logging becomes more significant as well as heap management since these operations modify the heap pages, and therefore, require log updates.

6. CONCLUSIONS

Recent studies emphasize that there is still a clear mismatch between what modern hardware offers and what OLTP systems need. Memory stalls dominate the overall execution time, and in turn, OLTP performance deteriorates and the underlying hardware remains largely under-utilized.

We conduct a detailed trace simulation on instruction and data misses modeling the memory hierarchy of one of the most commonly used hardware types. The experimental results link the most important memory-related stall types to software components in the storage manager, and quantify the effect of increasing the data size. More specifically, our results demonstrate that the L1 instruction misses are the main cause of the stalls even when working with large memory-resident data sets. The index probe operation, which is the most frequent routine for OLTP workloads, is the fundamental cause of both data and instruction misses coming from different levels of the cache hierarchy. The capacity misses coming from the B-tree, lock, and buffer pool management are the essential factor in the misses observed during an index probe.

To achieve a more graceful integration of hardware and software for OLTP systems, both of the layers should become more aware of each other. On the software side, reducing code complexity in the components mentioned above and designing more cache-friendly index structures are cru-

cial. On the hardware side, dedicating several close-by cores for specific transaction operations can help reducing the capacity misses as well as creating opportunities for hardware specialization.

7. REFERENCES

[1] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *MICRO*, pages 188–198, 2012.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *ISCA*, pages 3–14, 1998.

[3] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *ASPLOS*, pages 283–292, 2006.

[4] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. In *SIGMOD*, pages 157–168, 2002.

[5] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *ASPLOS*, pages 175–184, 2006.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012.

[7] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO*, pages 152–162, 2011.

[8] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-Conscious Frequent Pattern Mining on Modern and Emerging Processors. *The VLDB Journal*, 16(1):77–96, 2007.

[9] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008.

[10] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE TOCS*, 38(12):1612–1630, 1989.

[11] IBM Breaks Double Digit Performance Barrier With 10 Million Transactions Per Minute, 2010. http://www-03.ibm.com/press/us/en/pressrelease/32328.wss.

[12] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[13] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP Scalability Using Speculative Lock Inheritance. *PVLDB*, 2(1):479–489, 2009.

[14] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, pages 24–35, 2009.

[15] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A Scalable Approach to Logging. *PVLDB*, 3:681–692, 2010.

[16] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*, pages 15–26, 1998.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

[18] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *SIGMOD*, pages 371–380, 1992.

[19] SPARC Supercluster with 27 SPARC T3-4 Servers Demonstrates World Record Performance on TPC-C Benchmark, 2010. http://www.oracle.com/us/solutions/performance-scalability/t3-4-tpc-c-12210-bmark-190934.html.

[20] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. *PVLDB*, 3(1):928–939, 2010.

[21] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-Free Shared-Everything OLTP. *PVLDB*, 4(10):610–621, 2011.

[22] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *PVLDB*, 5(11):1447–1458, 2012.

[23] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA*, pages 155–164, 2001.

[24] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *ASPLOS*, pages 307–318, 1998.

[25] Shore-MT Official Website. http://diaswww.epfl.ch/shore-mt/.

[26] R. Stets, K. Gharachorloo, and L. Barroso. A Detailed Comparison of Two Transaction Processing Workloads. In *WWC*, pages 37–48, 2002.

[27] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[28] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored. In *EDBT*, pages 17–28, 2013.

[29] Transcation Processing Performance Council (TPC). http://www.tpc.org.

[30] VoltDB. VoltDB, 2012. http://www.voltdb.com.