From A to E: Analyzing TPC's OLTP Benchmarks

The obsolete, the ubiquitous, the unexplored

Pınar Tözün Ippokratis Pandis* Cansu Kaynak Djordje Jevdjic Anastasia Ailamaki

École Polytechnique Fédérale de Lausanne
Lausanne, VD, Switzerland

*IBM Almaden Research Center
San Jose, CA, USA

ABSTRACT

Introduced in 2007, TPC-E is the most recently standardized OLTP benchmark by TPC. Even though TPC-E has already been around for six years, it has not gained the popularity of its predecessor TPC-C: all the published results for TPC-E use a single database vendor's product. TPC-E is significantly different than its predecessors. Some of its distinguishing characteristics are the non-uniform input creation, longer-running and more complicated transactions, more difficult partitioning etc. These factors slow down the adoption of TPC-E. In turn, there is little knowledge in the community about how TPC-E behaves micro-architecturally and within the database engine.

To shed light on TPC-E, we implement it on top of a scalable open-source database engine, Shore-MT, and perform a workload characterization study, comparing it with the previous, much better known OLTP benchmarks of TPC: TPC-B and TPC-C. In parallel, we study the evolution of the OLTP benchmarks throughout the decades. Our results demonstrate that TPC-E exhibits similar micro-architectural behavior to TPC-B and TPC-C, even though it incurs less stall time and higher instructions per cycle. On the other hand, within the database engine it suffers more from logical lock contention. Therefore, we argue that, on the hardware side, TPC-E needs less aggressive processors. Whereas on the software side it can benefit from designs based on intra-transaction parallelism, logical partitioning, and optimistic concurrency control to minimize the effects of lock contention without introducing distributed transactions.

1. INTRODUCTION

For the past decades, the data management ecosystem and in turn the database and hardware markets have evolved primarily around two applications: online transaction processing (OLTP) and online analytical processing (OLAP). Transaction processing benchmarks are the gold standard for comparing products by different database and hardware vendors, and are regularly used for marketing purposes [16, 28]. For the last two decades, TPC-C [37] has been the most widely used OLTP benchmark by the majority of industry and academia. TPC-C consists of simple short-running transactions with frequent updates and less frequent index scans. On the other hand, the benchmark of choice for OLAP workloads is TPC-H [39]. TPC-H observes more complicated long-running read-only queries with frequent index and file scans. The data management stacks, from the database down to hardware, are typically optimized for these two extreme benchmarks.

In order to represent OLTP workloads more realistically, the Transaction Processing Performance Council (TPC) introduced the TPC-E benchmark [38] in 2007. TPC-E is an OLTP workload that includes transactions for real-time business intelligence combined with client-side requests. It acts in between a typical OLTP and an OLAP benchmark. The design decision for TPC-E was to create a sophisticated OLTP benchmark, having more complicated and longer transactions when compared to TPC-C, relying on the extensive use of non-primary indexes, observing data and access skew, applying integrity and referential constraints, and being less amenable to partitioning.

Both industry and academia are slow at adopting TPC-E. For example, even though the benchmark was standardized six years ago, all of the published results for TPC-E use the same database product (Microsoft SQL Server). Due to TPC-E's significant differences from the other benchmarks, it is not easy to extrapolate how systems perform when they run TPC-E (and TPC-E-like applications).

Existing experimental studies typically use database benchmarks other than TPC-E. Previous studies of OLTP and OLAP benchmarks, either micro-architectural [2, 5, 15, 22, 32, 33] or profiling [18, 20, 29, 30], provide valuable results. However, they fall short of explaining the behavior TPC-E is expected to exhibit. Recent work that analyzes TPC-E either focuses only on the I/O behavior [10, 21] or reports micro-architectural results on only one type of machine while running TPC-E on a commercial RDBMS and treating the database as a black-box [13]. To date, there is neither an analysis of the TPC-E benchmark on various hardware platforms nor a comprehensive breakdown of the execution time with respect to database engine components.

In this paper, we perform a detailed study of TPC-E. We characterize where it spends time within an open-source database engine and how it behaves micro-architecturally on two different hardware platforms, one in-order and one out-of-order machine. In parallel, we compare TPC-E to the well-known OLTP benchmarks and observe how TPC's

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

transactional benchmarks have evolved over the years. Then, we discuss what kind of changes in database and hardware systems can be more beneficial for such a workload. The contributions of our study are as follows:

- Our micro-architectural study demonstrates that TPC-E is actually very similar to the previous OLTP benchmarks in terms of its micro-architectural behavior. It highly suffers from L1 instruction misses and exhibits low instructions per cycle (IPC); IPC is smaller than one on a machine that has ability to execute four. Thus, we argue that TPC-E-like workloads need less aggressive processors with a lower instruction issue width on the hardware side. In addition, even though simultaneous multi-threading (SMT) hides some of the stalls caused by instruction misses and almost doubles the IPC, we need more effective solutions like intra-transaction parallelism [11, 29] or computation spreading [4, 9] to better utilize modern processor cores.
- Our profiling study reveals that, within the database engine, TPC-E spends 70% more time inside the lock manager compared to both TPC-B and TPC-C for a configuration with an orders of magnitude bigger database size. TPC-E's more complicated schema and transactions make it less straightforward to physically partition a TPC-E database to eliminate its locking overheads due to the significant number of distributed transactions such a design would cause. However, TPC-E can benefit from shared-everything designs that aim minimizing locking with logical [29] or physiological partitioning [30], or systems that rely on optimistic concurrency control [24] to improve system performance.

The rest of the paper is organized as follows. Section 2 briefly describes the previous transaction processing benchmarks standardized by TPC over the years and details TPC-E. Section 3 introduces Shore-Kits, which is a suite of OLTP benchmarks for Shore-MT and Section 4 describes our experimental methodology. Section 5 and Section 6 present the profiling and micro-architectural analysis respectively for TPC-E in comparison with TPC-B and TPC-C. Based on the analysis results, Section 7 discusses possible design optimizations both for upcoming hardware and storage managers, mainly while running TPC-E on top. Finally, Section 8 surveys the related work and Section 9 concludes.

2. EVOLUTION OF OLTP BENCHMARKS

Transaction processing benchmarks are the gold standard for DBMS performance evaluation and they are frequently used for marketing purposes. The Transaction Processing Performance Council (TPC) is a non-profit IT organization founded to define database benchmarks and disseminate objective, verifiable performance data to the industry. This section describes the four important database transaction processing benchmarks that have been used under the trademark of TPC and highlights how they have evolved over the years with each new benchmark.

2.1 The obsolete TPC-A and TPC-B

The first widely accepted database benchmark was formalized in 1985 [3]. That specification included three workloads, of which the "DebitCredit" stressed the database engine. The DebitCredit benchmark was an instant success. Soon database and hardware vendors started reporting extraordinary results, often achieved by removing key constraints from the specification. Therefore, in 1988 a consortium of analysts and hardware, operating system, and database vendors formed the Transaction Processing Performance Council in order to enforce some order in database benchmarking. Its first benchmark specification, TPC-A, essentially formalized the DebitCredit benchmark.

TPC-A is straightforward. It models deposits on and withdrawals from random bank accounts, with the associated double-entry accounting on a database that contains x `Branch`es, $10x$ `Teller`s, and $100,000x$ `Account`s. It also captures the entire system, including terminals and network. Transactions usually originate from their "home" `Branch`, but can go anywhere. Conflicts are possible requiring the system to recover occasionally from failed transactions. An important aspect of this benchmark is its scaling rule: for a result to be valid, the database size must be proportional to the reported throughput.

Simple though it maybe, the TPC-A benchmark highlighted the importance of quantifying the performance and correctness of different systems. Early benchmarking showed vast performance differences among different vendors (400x), as well as exposing serious bugs, which had been lurked and undiscovered, for many years in mature products.

TPC's second benchmark, TPC-B [35], is very similar to TPC-A, but eliminates the network and terminal handling to create a database engine stress test. Like TPC-A, the TPC-B database contains four tables: `Branch`, `Teller`, `Account`, and `History`. These tables are accessed in a double-entry accounting style as customers make deposits on and withdrawals from various tellers. The benchmark consists of a single transaction, `AccountUpdate`, which simply updates one record in the `Branch`, `Teller`, and `Account` tables while appending a record to the `History` table. Therefore, it is a very update-heavy transaction that stresses the transaction processing engine; especially the logging and concurrency control modules. Due to the similarities between TPC-A and TPC-B, for the rest of our study we use TPC-B alone.

2.2 The ubiquitous TPC-C

For its third benchmark specification, TPC-C [37], TPC moves away from banking to commerce. TPC-C models an online transaction processing database for a wholesale supplier. The transactions follow customer orders from initial creation to final delivery and payment.

A TPC-C database consists of nine tables in total where one of them has fixed size (*Fixed*), four of them scale proportionally with the number of `Warehouse`s (*Scaling*), and four of them might change size, mostly grow, due to insert and delete operations (*Growing*). Thereby, compared to TPC-B, TPC-C offers a more complex database schema; where the TPC-B schema can be represented as a tree with only four nodes, the TPC-C schema is a directed acyclic graph with nine nodes.

Like the database schema, the TPC-C transactions are also more complex. The benchmark combines the five transactions listed below in a transaction mix at frequencies given in parenthesis:

- `NewOrder` **(45%)** inserts a new sales order to the database. It is a medium-weight transaction with a 1% failure rate due to invalid inputs.

Table 1: The TPC-E transactions

Transaction	Weight	Access	Category	Frames Executed	% in Mix
BrokerVolume	Mid to Heavy	RO	BI	1 (out of 1)	4.9
CustomerPosition	Mid to Heavy	RO	CI	2/3 (out of 3)	13
MarketFeed	Medium	RW	MT	1 (out of 1)	1
MarketWatch	Medium	RO	CI	1 (out of 1)	18
SecurityDetail	Medium	RO	CI	1 (out of 1)	14
TradeLookup	Medium	RO	BI/CI	1 (out of 4)	8
TradeOrder	Heavy	RW	CI	2/5/6 (out of 6)	10.1
TradeResult	Heavy	RW	MT	5/6 (out of 6)	10
TradeStatus	Light	RO	CI	1 (out of 1)	19
TradeUpdate	Medium	RW	BI/CI	1 (out of 3)	2
BI: Brokerage Initiated, CI: Customer Initiated, MT: Market Triggered					

- `Payment` **(43%)** is a short transaction, very similar to the `AccountUpdate` transaction of TPC-B, which makes a payment on an existing order.
- `OrderStatus` **(4%)** is a read-only transaction that computes the shipping status and the line items of an order.
- `Delivery` **(4%)** is the largest and the most contentious update transaction. It selects the oldest undelivered orders for each warehouse and marks them as delivered.
- `StockLevel` **(4%)** is also a read-only transaction. It joins on average 200 order line items with their corresponding stock entries in order to produce a report.

The specification also lays out strict requirements about response time, consistency, and recovery in the system, and brings back the testing an end-to-end system that includes network and terminal handling.

TPC-C stresses the entire stack (database system, operating system, and hardware) in several ways. First, it mixes short and long, read-only and update-intensive transactions, exercising a wider variety of features and situations than the TPC-B benchmark. In addition, the benchmark has major hotspots, partly due to the way transactions access the `Warehouse` table and partly due to how the `Delivery` transaction is designed. The resulting contention and deadlocks stress the system's concurrency control mechanisms. Finally, the database grows throughout the benchmark run; not just because of the append-only `History` table as in TPC-B, but also because of the insert and delete operations on different tables, stressing code paths that the previous TPC-B benchmark did not reach.

TPC-C has been the most popular OLTP benchmark for over twenty years. Major database vendors have published results on TPC's website, and on several occasions it is used for marketing purposes [16, 28].

2.3 The unexplored TPC-E

To represent more realistically real-life OLTP workloads, TPC presented TPC-E [38] as an alternative to the dominant TPC-C. In this subsection, we give an overview of TPC-E while pointing out its differences from TPC-C.

2.3.1 Model

TPC-E models a brokerage house. The database tables keep information about the customers, brokers, and market. The transactions simulate a workload where either the customers initiate requests to the brokerage house (*customer initiated transactions*) or the market sends ticker feeds or trade results to the brokerage house (*market-triggered transactions*). The brokerage house responds to the customers, checks the orders to decide whether to submit them or not, submits the related brokerage requests (*brokerage initiated transactions*), and analyzes or updates the database. One could say that TPC-E represents a more complicated business model compared to TPC-C.

2.3.2 Database

TPC-E has more tables than TPC-C; thirty-three tables instead of nine. Nine of TPC-E's tables are of *Fixed* size, sixteen are *Scaling* based on the number of `Customer`s, and eight are *Growing*. However, the growth rate of the *Growing* tables varies and in general it is greater than the growth rates of the *Growing* tables in TPC-C. In addition, the TPC-E tables are populated with pseudo-real data and exhibit data skew. On the contrary, TPC-C tables have randomly generated data that face a low degree of skew.

The scaling factor determines the number of `Branch`es in TPC-B and the number of `Warehouse`s in TPC-C. TPC-E has a scaling factor that controls the number of `Customer`s in the database. But, unlike TPC-B and TPC-C, where a single scaling factor (via the number of `Branch`es and `Warehouse`s) is the only parameter that determines the initial size of the database, TPC-E has two additional parameters that affect the initial database size. In particular, the parameters called *working days* and *scaling factor* control the cardinality of the `Trade` table and in turn all the other *Growing* tables in TPC-E.

TPC-E also has a *Growing* table, `Trade_Request`, that right after database population starts as an empty table and then grows. Neither TPC-B nor TPC-C have empty tables after the initial database population.

2.3.3 Transactions

TPC-E contains twelve transactions in total, which are shown in Table 1. Only ten of the transactions belong to the regular transaction mix. Two of them, `DataMaintenance` and `TradeCleanup`, get executed separately. `DataMaintenance` is executed periodically, every minute, alongside with the transaction mix, whereas `TradeCleanup` needs to be executed before each run if one wants to cleanup the submitted or pending trades from a previous run in order to restore the initial database state. In TPC-C, all of the five transactions

Table 2: Evolution of TPC's OLTP benchmarks

		TPC-A	TPC-B	TPC-C	TPC-E
First release		Nov 1989	Aug 1990	Aug 1992	Feb 2007
Last update		Jun 1994	Jun 1994	Feb 2010	Jun 2010
Business model		Banking	Banking	Wholesale supplier	Brokerage house
Tables	Fixed	0	0	1	9
	Scaling	3	3	4	16
	Growing	1	1	4	8
	Total	4	4	9	33
Transactions	RW	1	1	3	6
	RO	0	0	2	6
Transaction Mix %	RW	100%	100%	92%	23.1%
	RO	0%	0%	8%	76.9%
Transactions using secondary indexes		None	None	2	10
Data population		Random	Random	Random	Pseudo-real

are included in the transaction mix.

The TPC-E transactions consist of *frames*, which are parts of a long transaction with a distinctive task. For some transactions only a subset of their frames are executed depending on the input values or whether they are initiated by a customer or brokerage; like in `TradeLookup` and `TradeUpdate`. TPC-C does not contain as complicated and long transactions. All transactions in TPC-C have only one frame.

One significant distinction of TPC-E from its predecessors is the majority of the transactions in the mix are Read-Only (RO). That is, in TPC-E around 75% of the transactions executed are read-only, whereas TPC-C has 92% Read-Write (RW) transactions in the mix.

Another distinction of TPC-E is that its transaction mix enforces dependencies among some of the transactions. More specifically, the market-triggered transactions (`TradeResult` and `MarketFeed`) require `TradeOrder` transactions to submit input for them. Therefore, they cannot be executed independently from the transaction mix. In TPC-C none of the transactions have such dependencies.

TPC-E specification also introduces skew in transaction inputs, harness control measures within the transactions, and checks for referential integrity constraints, which do not exist in TPC-C. Moreover, for high performance, TPC-E needs to perform lookups and scans through non-primary indexes in almost all of its transactions (ten out of twelve), whereas TPC-C uses secondary indexes in only two of its transactions.

Overall, TPC-E is a much more sophisticated OLTP benchmark compared to all its predecessors and therefore, it offers a more interesting and mature environment for testing OLTP engines. On the other hand, it is also harder to adopt for people from both industry and academia, which have been optimizing their systems mainly based on TPC-C for the last twenty years.

2.4 The evolution summary

Table 2 summarizes the high-level comparison of the four OLTP benchmarks of TPC, which we detailed above. What we can conclude from this section and Table 2 is that with each benchmark TPC standardized, we see a significant complexity increase, which is driven by the facts listed below:

- A more sophisticated business model.
- A larger variety in terms of transaction types.
- Longer-running and less deterministic transactions, causing longer and less predictable instruction streams.
- Increase in the number of read-only transactions that need to be run together with update-heavy ones.
- Increase in the number of scan operations and dependency on the secondary indexes, which in turn makes physical database partitioning less effective.
- More fundamental stress within the storage manager and exploration of an increased number of code-paths.

The above items are going to be crucial while explaining the behavior of these workloads within a storage manager and micro-architecturally.

3. SHORE-KITS: BENCHMARKS ON TOP OF SHORE-MT

Shore-MT [19] is an enhanced version of the SHORE storage manager [8], whose micro-architectural behavior is very close to the commercial systems [1]. Shore-MT adds a multithreaded storage manager kernel to SHORE and is particularly developed to adapt SHORE to multicore era, mainly by focusing on eliminating the scalability bottlenecks when running on multicore hardware. Today, Shore-MT is one the most scalable open-source shared-everything storage managers within a single database node. It has been used in various research projects as a test-bed both by the team who develops and maintains it [18, 20, 29, 30, 31] and by other well-known teams in the database and computer architecture communities [4, 14].

In order to study the behavior and challenges the standardized OLTP benchmarks pose on modern storage managers, we implement them on top of Shore-MT and distribute them as a suite of database benchmarks, called Shore-Kits. In other words, Shore-Kits[1] is an open-source suite of OLTP benchmarks for the Shore-MT storage manager.

[1]Available at https://bitbucket.org/shoremt

Since Shore-MT does not have an SQL front end, a query parser, and an optimizer, the benchmarks are implemented in C++ using direct calls to Shore-MT's storage manager API, which is linked as a static library to the executable. With some programming effort and code refactoring, one can port Shore-Kits to other storage managers by changing the API calls to match the target storage manager's API.

We implemented TPC-E using the query plans taken from a TPC-E implementation of a major database vendor. As for the index decisions, we initially adapted the indexes from the same kit. Later, however, we had to change some of the indexes in order to optimize performance when running on top of Shore-MT. For example, Shore-MT's API allows Shore-Kits to use only unclustered indexes, whereas the kit of the commercial database uses clustered ones for the primary indexes. Therefore, the optimal index decisions varied between Shore-Kits and the kit of the commercial database. Due to its large number of tables and longer and more complicated transactions, TPC-E was by far the most difficult benchmark implemented in Shore-Kits.

TPC-E stresses Shore-MT in ways previous benchmarks do not. It pinpointed code-paths, exposing previously undetected bugs and performance bottlenecks. Therefore, it helped us to further improve Shore-MT. For example, Shore-MT had implementation of forward and backward index scans. But the backward index scans were disabled, because they were causing large number of deadlocks in some workloads. Debugging and re-enabling backward index scans in Shore-MT improved performance of TPC-E by three orders of magnitude on an Intel server.

4. EXPERIMENTAL METHODOLOGY

We used two servers for our experiments: (1) a Sun UltraSPARC T5220 server with one socket containing eight in-order cores, where each core has support for eight hardware contexts and is clocked at 1.4GHz, running Solaris 10, and (2) a server with two Intel Xeon X5660 processors each with six out-of-order processor cores running Ubuntu 10.04 with Linux kernel version 2.6.32. Table 3 lists the characteristics of each processor in detail. The diversity and degree of hardware parallelism on these systems make them good candidates for this study to reflect the behavior of our workloads on various types of modern hardware.

We use memory-resident databases for our experiments and flush the log to RAM due to not having a suitably fast I/O sub-system. A configuration that allows I/O in our infrastructure might cause an unreasonably slow and highly suboptimal OLTP system, and therefore, unrealistic micro-architectural conclusions.

On the Intel machine, we experiment with two cases; when hyper-threading (HT) is off and when it is on. When hyper-threading is on, the Intel machine supports two hardware contexts running at the same time on one core to be able to overlap the stall time of one of the threads with the execution of the other. This property is analogous to the simultaneous multi-threading (SMT) support in the SPARC machine where each core has support for eight hardware contexts by default, which is actually one of the main design principles of the UltraSPARC T2 architecture.

We chose the most optimal configuration options we determined empirically for all the benchmarks running on top of Shore-MT to make sure that we run them without any obvious scalability bottlenecks and better utilize the hardware resources. In TPC-B we pad the records of `Branch` and `Teller` tables so that a single database page only has a single record. This minimizes false sharing of database pages and avoids latching contention, which can be a fundamental bottleneck for typical shared-everything architectures [30]. We also enable Speculative Lock Inheritance (SLI) [18] and logging optimizations from Aether [20] to reduce the bottlenecks coming from the lock and log managers, respectively, for the benchmarks that benefit from these techniques.

Table 3: Server Properties

Server	UltraSPARC T2	Intel Xeon X5660
#Sockets	1	2
#Cores per Socket	8 (in-order)	6 (OoO)
#HW Contexts	64	24
Clock Speed	1.40GHz	2.80GHz
Memory	64GB	48GB
L3 (shared) access latency	-	12MB 29 cycles
L2 (shared) access latency	4MB 20 cycles	-
L2 (per core) access latency	-	256KB 6 cycles
L1-I (per core) access latency	16KB 3 cycles	32KB 4 cycles
L1-D (per core) access latency	8KB 3 cycles	32KB 4 cycles
OS	SunOS 5.10 Generic_ 141414-10	Ubuntu 10.04 with Linux kernel 2.6.32

Furthermore, for TPC-B and TPC-C we spread the requests based on the primary key of the `Branch` and `Warehouse` tables, respectively, to reduce logical lock contention. In order to do that, we picked scaling factors that are equal to the number of hardware contexts available on the machine a specific experiment is run on, since the scaling factor is equal to the number of `Branch`es in TPC-B and `Warehouse`s in TPC-C. In other words, on the Intel machine we picked a scaling factor of 12 and 24 when hyper-threading is disabled and enabled, respectively, and on the SPARC machine we picked a scaling factor of 64. Unfortunately, for TPC-E, it is not straightforward how to spread the requests due to its more complex schema and transactions that do not have correlation based on any primary key column for the majority of the database tables. To be able to run an in-memory database, we picked a database size that contains 1000 customers for TPC-E. We set the *working days* and *scaling factor* parameters to 300 and 500, respectively, which are the default values for these parameters in the TPC-E specification.

Before taking any measurements, we start with a newly populated database, make each worker thread in the system execute 1000 transactions to warm-up the caches, and then perform a one-minute run. The tools used to collect the hardware counter values and profiling results during these runs are mentioned in the related sections.

5. PROFILING ANALYSIS

In order to further understand the high-level characteristics of each benchmark, firstly, we report statistical infor-

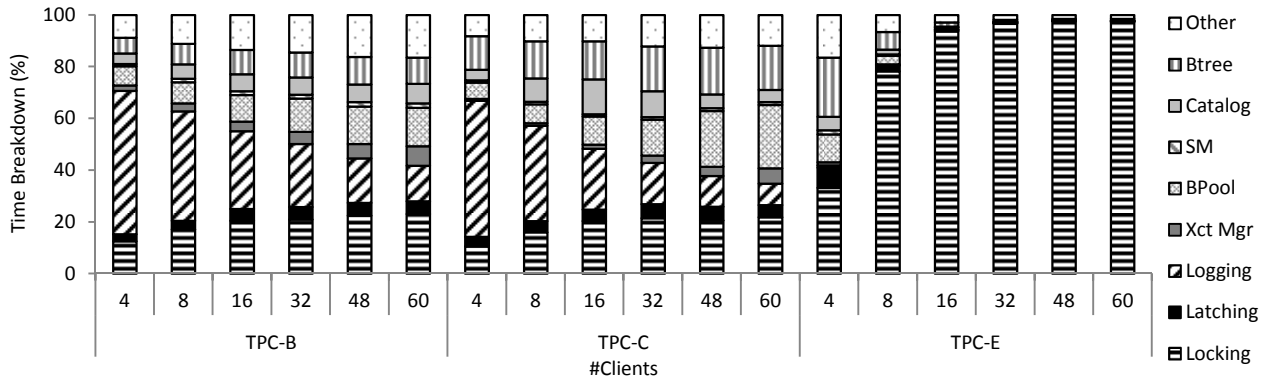

Figure 1: Time breakdown as the machine load increases on UltraSPARC T2.

mation collected from the storage manager in Section 5.1. Then, in Section 5.2, our profiling analysis identifies the components of the storage manager each benchmark spends the most time in.

5.1 High-level analysis

Table 4 contains the high-level statistics of each benchmark to further highlight the changes in complexity with each OLTP benchmark standardized by TPC. These statistics are independent of the underlying hardware. We chose a scaling factor of one for each benchmark in this part of the analysis. This corresponds to one `Branch` in TPC-B, one `Warehouse` in TPC-C, and one-thousand `Customer`s in TPC-E. For the initial database, we measure the number of records each benchmark has and how many pages it uses in Shore-MT, which uses 8KB pages by default. Then, we use the existing statistic measurements within Shore-MT to see how many records, locks, and pages on average a transaction accesses for each benchmark while performing a run with one worker thread executing transactions.

As expected, Table 4 re-emphasizes the complexity increase from TPC-B to TPC-E. TPC-E has several orders of magnitude more records per scaling factor compared to TPC-B and TPC-C, and a much larger database size as the total number of heap and index pages indicates. TPC-B only touches one record per table, hence it accesses few database locks and pages. TPC-C accesses almost ten times the records TPC-B accesses per transaction in its transaction mix, increasing the number of locks and database pages it accesses as well. Finally, TPC-E performs around four times the record accesses of TPC-C, which is also reflected in the higher number of row-level locks it has to acquire. However, the total number of locks acquired does not increase accordingly since Shore-MT escalates to higher-level locking from row-level locking when a single transaction accesses more than a threshold of records (the default value is twenty-five in Shore-MT).

Table 4 reports two values for the average number of pages accessed in a transaction; the unique number of pages accessed and the total number of pages accessed, which is also the number of times a page is requested from the buffer pool. Such a separation reveals that even though TPC-E accesses more than twice the index pages TPC-C does, the number of unique index page accesses is the same for both workloads. The main reason for this is TPC-E's extensive index scans. TPC-C does not re-access most of the index pages it touches, while TPC-E does this very frequently for the index leaf pages during its index scans; it sequentially reads an index leaf page and hence frequently reuses that page. This results in TPC-E exhibiting lower L1 data cache miss rates as Section 6.1.3 and Section 6.2 show.

Table 4: High-level statistics of each benchmark per scaling factor 1

	TPC-B	TPC-C	TPC-E
# records	~ 10K	~ 600K	~ 117M
# heap pages	147	~ 12K	~ 1M
# index pages	91	~ 6K	~ 1M
Average per xct			
# records accessed	4	36	149
# row-level locks	10	54	171
# higher-level locks	10	36	69
# heap pages accessed (U)	4	23	40
# index pages accessed (U)	4	33	33
# heap pages accessed	7	49	125
# index pages accessed	4	90	211
K: thousand, M: million, U: unique			

5.2 Time breakdown

To get accurate time breakdowns within the storage manager, we use DTrace [7] on the SPARC machine. Figure 1 presents the results of the profiling as we increase the machine utilization, i.e., as we run more clients in the system.

Figure 1 highlights that the lock manager is one of the components the OLTP benchmarks spend most of their time in within a shared-everything database management system. The lock manager becomes the main bottleneck especially for TPC-E, making it unable to utilize more than eight hardware contexts on this machine, while both TPC-B and TPC-C are able to almost fully utilize the machine with smaller database sizes.

Logging is the next problematic component for TPC-B and TPC-C. It becomes, however, less significant as we increase the system utilization since we adopt the logging optimizations of [20] that benefit from combining logging requests as the number of clients in the system increases. *Btree* and *BPool* (buffer-pool) come after *Locking* and *Logging*, since a transaction's execution is highly dependent on

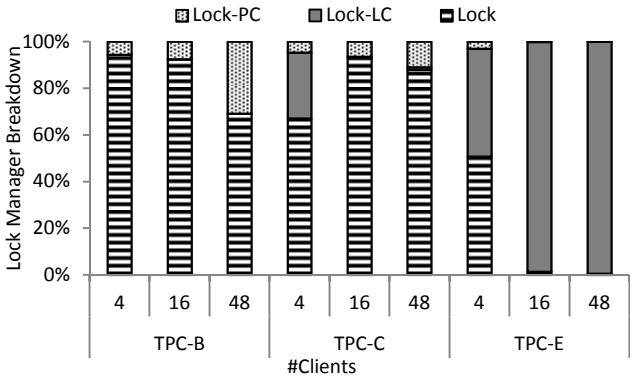

Figure 2: Time breakdown in the lock manager as the machine load increases on UltraSPARC T2.

its index operations. The rest of the major components of a storage manager are *Catalog* (metadata manager), *SM* (storage manager API functionality), *Xct Mgr* (transaction manager), and *Latching*; in which none of the workloads spends a major part of their execution time.

Figure 2 focuses on the time spent inside the lock manager and shows the time breakdown of sub-categories: Physical lock contention, *Lock-PC*, represents the time spent while waiting to acquire the element guarding a particular record or table lock. Logical lock contention, *Lock-LC*, represents the time spent until a record or table lock is granted after the lock request is appended to the list of requests for this lock. Finally, locking, *Lock*, is the time spent on performing the locking operation aside from the waiting time.

TPC-E mainly suffers from logical lock contention (Lock-LC) even though we use a larger database size for it compared to TPC-B and TPC-C. There are three main reasons for this outcome: (1) TPC-E observes data and access skew, turning some of the data regions into hotspots (e.g., `Last_Trade` table); (2) TPC-E transactions acquire on average more locks since they access a larger number of database records; and (3) TPC-E transactions hold the locks they acquire for a longer duration since they are more complicated, longer running, and scan-heavy transactions. TPC-B and TPC-C, on the other hand, do not suffer from logical lock contention since the system can properly spread the requests and SLI [18] prevents physical lock contention from becoming problematic, leaving only the actual locking operation as the main time-consuming component within the lock manager.

However, as we will see in Table 5, the lock contention is not as problematic when we run TPC-E on the Intel machine, which has faster processors than the SPARC machine. The faster the processor, the faster the lock acquisitions and releases are, and hence, the less time is spent on lock contention. We come across this fact also when we run TPC-B. When two threads want to access the same `Branch` in a TPC-B database, they first acquire a read lock on the wanted `Branch` during the index probe according to ARIES/IM [26] (the default concurrency control scheme in Shore-MT). Later, when they want to upgrade their read locks to exclusive ones to update the `Branch`, they both wait for each other and they deadlock. While on the SPARC machine we observe such deadlocks, TPC-B runs without deadlocks on the Intel machine due to faster locking operations. Switching to ARIES/KVL [25], which has stricter concurrency control rules than ARIES/IM, makes this type of deadlocks disappear on the SPARC machine as well.

Table 5: Number of worker threads used for each benchmark on the two machines

Server	UltraSPARC T2	Intel Xeon X5660	
		No HT	HT
TPC-B	48	10	18
TPC-C	60	10	18
TPC-E	4	12	24

6. MICRO-ARCHITECTURAL ANALYSIS

While performing a micro-architectural analysis for the OLTP benchmarks, we try to answer the following questions: (1) Where do CPU cycles go on different types of modern hardware? Are they wasted on memory stalls or used to retire an instruction?, (2) Do stalls happen mainly due to instructions or data?, (3) How important are the instruction and data miss rates?, (4) How much instruction-level (ILP) and memory-level (MLP) parallelism do OLTP benchmarks exhibit?, and (5) What is the effect of simultaneous multi-threading (or hyper-threading)?

All the numbers reported in this section were obtained when the workloads have their peak performance on the corresponding server with their optimal configuration on Shore-MT. Table 5 shows the number of worker threads executing transactions in the system when the peak throughput is achieved for each workload on each server. Adding more worker threads to the system on top of the numbers reported in Table 5 causes degradation in throughput, either due to contention on shared records and storage manager objects or over-saturation of the machine being used.

6.1 OLTP on an out-of-order processor

This section presents micro-architectural results from the Intel Xeon X5660 processors. We use VTune [17], which provides an API to ease the use of the hardware counters on this machine. We emphasize that the execution time breakdown on a superscalar out-of-order (OoO) processor cannot be precise due to overlapping of different execution components [12]. However, considering the low IPC of the workloads we are experimenting with (Section 6.1.4), we can assume that not much of work is overlapped. Nevertheless, we draw the execution cycles that can be overlapped side-by-side rather than on top of each other.

Intel Xeon X5660 processors support hyper-threading, running two hardware contexts on one core at the same time. The goal of hyper-threading is to overlap the stall time of one thread with the execution of another. In the following subsections, for each experiment we present results when hyper-threading is disabled and when it is enabled.

6.1.1 Execution time breakdown

Figure 3 shows the breakdown of the execution cycles into busy and stall time for the three benchmarks. We count the cycles in which at least one instruction is retired as *busy* and where no instruction is retired as *stalled*.

In Figure 3, we see that more than half of the execution time is spent on stalls for all of the OLTP benchmarks. While TPC-B and TPC-C show very similar behavior in

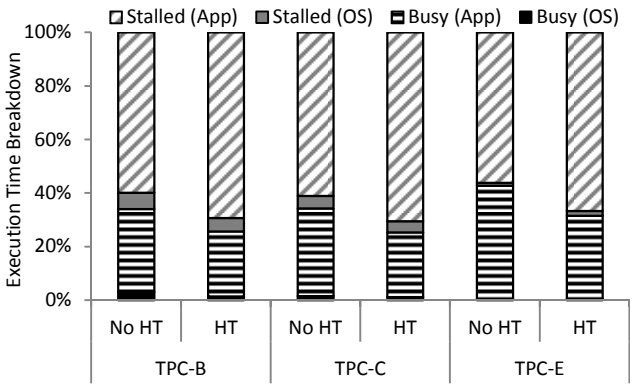

Figure 3: Execution time breakdown for three OLTP benchmarks on an OoO processor with and without hyper-threading.

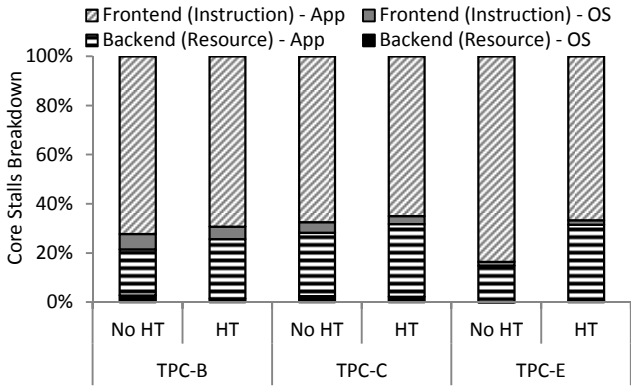

Figure 4: Core stalls breakdown for three OLTP benchmarks on an OoO processor with and without hyper-threading.

terms of the percentage of *busy* and *stalled* cycles, TPC-E seems to observe fewer *stalled* cycles during the overall execution. This behavior results in a higher IPC value for TPC-E (see Section 6.1.4). As expected, when hyper-threading is enabled, the *stalled* cycles increase in the overall execution time since two threads instead of one share the private L1 and L2 caches, evicting each other's data and instructions from the cache, thus, causing more cache misses.

Figure 3 also breaks the execution time into time spent on the operating system operations (*OS*) and application itself (*App*); and it demonstrates that for our configuration, the *OS* does not contribute much to the overall execution time.

6.1.2 Core stalls

As presented in the previous section, stalls dominate the total execution time of OLTP benchmarks. The estimated breakdown of these stalls into resource, which also includes data, and instruction stalls are given in Figure 4. We account resource stalls within a core, mainly stemming from the re-order buffer (ROB) being full, as *backend/resource* stalls while the remaining stalls as *frontend/instruction* stalls. We, again, separate OS and application stalls even though OS does not contribute significantly to the total stall time.

As Figure 4 demonstrates, the main cause of core stalls is the frontend stalls for the OLTP benchmarks. In other words, a core spends most of its execution cycles waiting for instructions, since it cannot find them in its private L1 instruction cache. The percentage of the frontend stalls is higher for TPC-E compared to both TPC-B and TPC-C. We link this behavior to lower data miss rate of TPC-E (see Section 6.1.3), which increases the percentage of stalls for instructions.

In addition, hyper-threading increases the percentage of the backend stalls. Two threads sharing the resources of one core with hyper-threading can increase the hit rate of the instruction cache more than the data cache, because transactions tend to share more instructions than data [4].

6.1.3 Data and instruction misses

Figure 5 shows the number of misses per k-instructions on the left-hand side and the estimated number of cycles spent on these misses on the right-hand side. As we mentioned before, we demonstrate the cycles spent on various cache misses side-by-side rather than on top of each other because of the unknown overlapping cycles for these misses. We categorize the cache misses as L1 instruction cache misses (*L1I*), L2 instruction misses (*L2I*), L1 data cache misses (*L1D*), L2 data misses (*L2D*), and L3 or last-level cache misses (*LLC*). For stall cycles due to cache misses, we use the expected penalty for that particular miss on the machine being used. For LLC misses, we average the penalty for going to local memory and remote memory.

What we observe is that L1 instruction cache misses dominate both the total number of misses and the total number of cycles spent on those misses for all of the OLTP benchmarks. As mentioned in Section 6.1.1, enabling hyper-threading increases the total number of misses in general due to more threads sharing the shared cache resources.

TPC-E exhibits ∼35% fewer data misses and almost the same number of instruction misses, regardless of its longer running and more complicated transactions. Since it performs more scan operations, TPC-E can reuse the cache lines for data and instructions it needs more often.

6.1.4 Instruction- and memory-level parallelism

Finally, Figure 6 shows how many instructions per cycle (IPC) these OLTP benchmarks can execute per core on the left-hand side and how many long-latency misses (L2 miss) can be overlapped (MLP) on the right-hand side.

An Intel Xeon X5660 processor has the ability to retire four instructions per cycle. However, by looking at Figure 6, we see that OLTP benchmarks can hardly retire even one instruction per cycle even though enabling hyper-threading provides some benefit. Overall, as the complexity of the benchmark increases, going from TPC-B to TPC-E, the IPC also increases. As we also mentioned in Section 6.1.2, it is expected that TPC-E has a higher IPC value since it spends less of its execution time on stall cycles compared to the other two workloads. Higher IPC stems from TPC-E observing fewer L1 data misses (Section 6.1.3) because of its frequent scan operations.

From the MLP values given in Figure 6, we also conclude that OLTP benchmarks do not exhibit high MLP. Even though there are 48-entry load-store queues in this processor, OLTP benchmarks do not have more than 2.7 outstanding long-latency misses even when hyper-threading is enabled. While TPC-B and TPC-C observe very similar MLP values, TPC-E exhibits less memory-level parallelism.

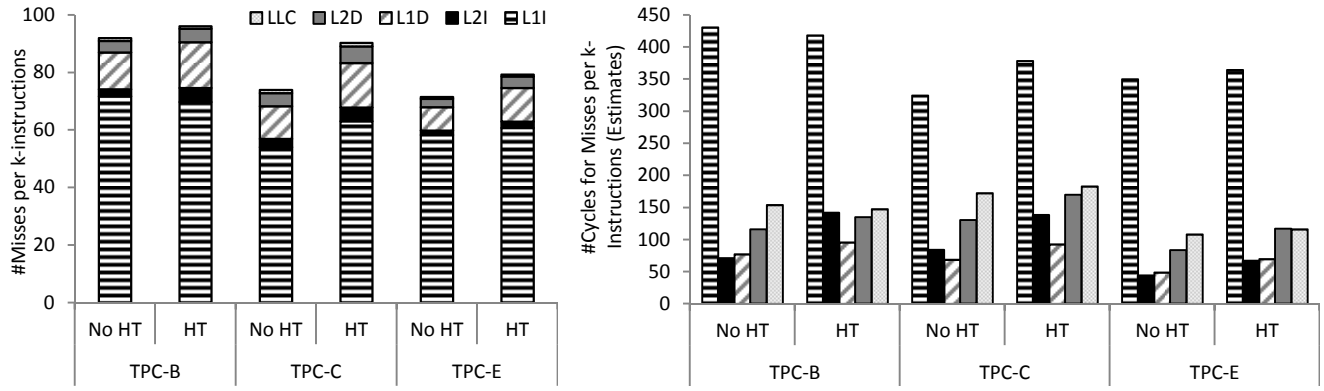

Figure 5: Number of misses per k-instructions for three OLTP benchmarks on an OoO processor with and without hyper-threading and the estimated number of cycles spent on these misses.

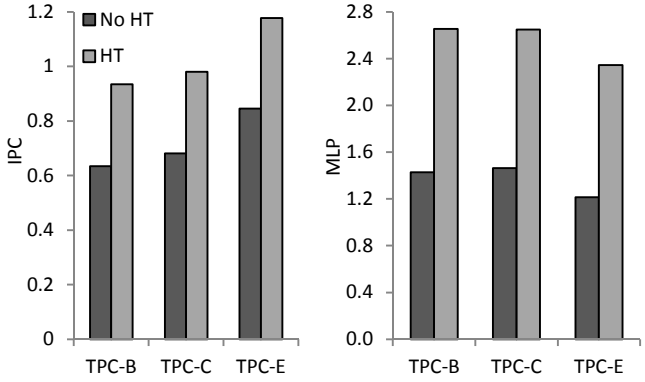

Figure 6: Instructions committed per cycle and memory-level parallelism on an OoO processor with and without hyper-threading.

6.2 OLTP on an in-order processor

This section presents micro-architectural results from the Sun UltraSPARC T5220 server. We used the hardware counters on this machine through the `cputrack` command [27], which allows us to count various types of cache misses and number of instructions executed by each thread.

UltraSPARC T2 is an in-order processor that supports simultaneous multi-threading. A core provides support for eight hardware contexts and collocates two hardware contexts in the pipeline in one cycle. Therefore, each of these hardware contexts uses one cycle in every four cycles, aiming to overlap the stall time of other hardware contexts.

Figure 7 shows the number of misses per k-instructions on the left-hand side and the estimated number of cycles spent on these misses on the right-hand side as in Figure 5. On this processor, we also cannot infer the overlapped operations and, as in Figure 5, we draw the execution cycles that can be overlapped side-by-side rather than on top of each other. We report L1 instruction cache misses (*L1I*), L2 instruction misses (*L2I*), L1 data cache misses (*L1D*), and L2 data misses (*L2D*). For stall cycles due to misses, we use the expected penalty for that particular miss on this machine.

Similar to the Intel machine, the main source of misses and stall cycles are also L1 instruction cache misses as Figure 7 shows. On the other hand, the last-level cache (L2) maintains almost all of the instructions for these workloads running on Shore-MT. Due to having smaller L1 data caches and more hardware contexts using the same private L1 cache in a core, L1 data cache misses contribute to a bigger portion of the total stall cycles compared to the Intel machine.

The comparison among the three benchmarks in terms of misses look similar to the comparison we have on the Intel machine (Figure 5). The instruction miss numbers are very close to each other for all the workloads and TPC-E has 50% fewer data misses compared to TPC-B and TPC-C.

Figure 8 shows the IPC values for the three OLTP benchmarks running on UltraSPARC T2. Considering that this is an in-order machine, being able to execute instructions from two hardware contexts in a cycle, the IPC being higher than one shows a more effective use of the hardware resources compared to the Intel machine. While, on the Intel machine, OLTP benchmarks can hardly leverage less than half of the instruction issue width, on SPARC, they can utilize more than half of it.

7. DISCUSSION

In this section we summarize the highlights of our experimental study and discuss the optimal ways of executing OLTP benchmarks, mainly focusing on TPC-E.

Looking at the high-level description and statistics for each benchmark, we see that with each new OLTP benchmark standardized by TPC, we have a significant increase in complexity compared to the previous ones. Moreover, observing our time breakdown results from Section 5.2 and previous studies [18, 20, 29, 30], each benchmark stresses different parts of the storage manager in different ways. However, regardless of these differences, micro-architecturally, all the OLTP benchmarks that exist today observe very similar behavior (Section 6).

As our micro-architectural analysis show, TPC-E a has higher IPC, observes lower miss rates, and spends less of its execution time on memory stalls compared to TPC-B and TPC-C. However, the fact that OLTP benchmarks commonly observe low IPC, spend most of their execution time on memory stalls, and mainly suffer from L1 instruction cache misses still remains. Going from an aggressive out-of-order processor to an in-order processor, does not change the micro-architectural characteristics of the OLTP benchmarks much. However, we observe that simultaneous multi-

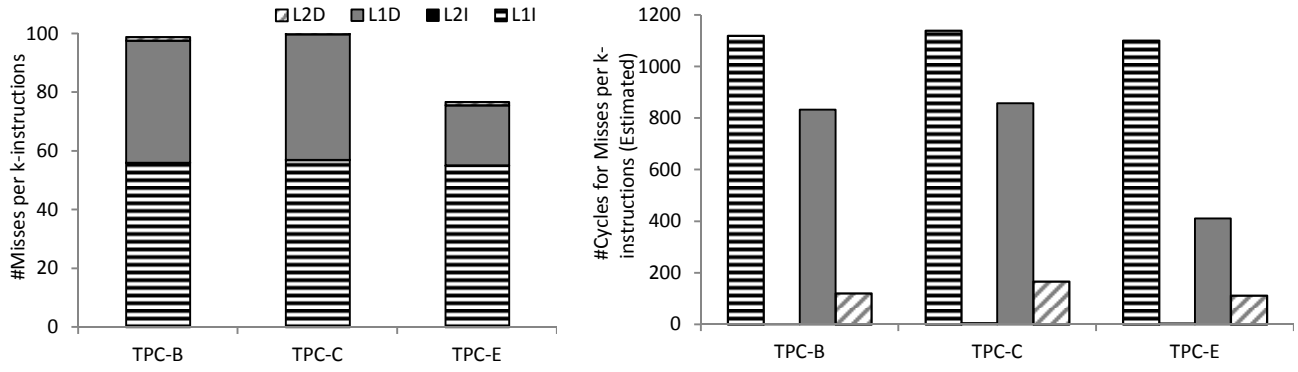

Figure 7: Number of misses per k-instructions for three OLTP benchmarks on an in-order processor with simultaneous multi-threading and the estimated number of cycles spent on these misses.

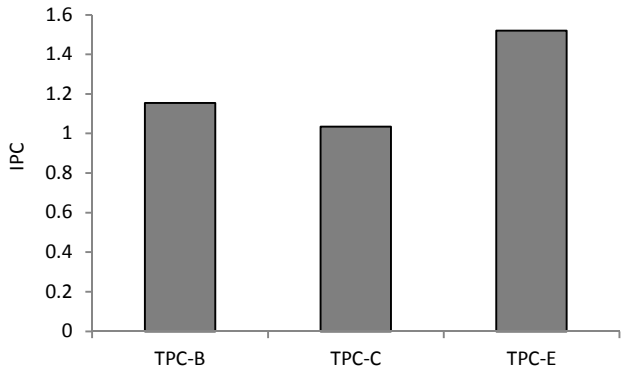

Figure 8: Instructions committed per cycle on an in-order processor with simultaneous multi-threading.

threading (or hyper-threading) helps to overlap the stall time caused by cache misses to some extent.

By looking at the time TPC-E spends inside the lock manager, the natural choice would be to partition the database and deploy a shared-nothing design for it. Even though for TPC-B- and TPC-C-like database schemas, this would work very well [34], for TPC-E such a design would cause a lot of distributed transactions. There are two main reasons for this: (1) Due to its complex schema, not all the TPC-E tables can be correlated with a single database column like the `Branch ID` in TPC-B or `Warehouse ID` in TPC-C. (2) The TPC-E transactions access a lot of database records from various tables and perform frequent index scans by using secondary indexes. Therefore, it is not clear based on which columns we should partition TPC-E tables in a way to minimize distributed transactions when we deploy a shared-nothing design.

On the other hand, a shared-everything design based on logical or physiological partitioning like in DORA [29] or PLP [30], respectively, might be more beneficial especially for TPC-E-like workloads. Such designs successfully minimize locking and latching overheads within the storage manager and they do not suffer from distributed transactions like in a shared-nothing design. In addition, optimistic and multiversion concurrency control schemes [6, 24] may especially help TPC-E-like read-heavy workloads to improve concurrency by avoiding blocking at the time of a potential conflict and rather lazily performing checks at commit time.

Considering that L1 instruction cache misses dominate the total number of cache misses, techniques that involve several cores to execute a transaction or exploit common instructions both within the same transaction and across different transactions would be very helpful for OLTP. While using several cores creates an aggregate L1 instruction cache capacity for a transaction, being able to reuse common instructions reduces the need to re-fetch an instruction over and over.

Software-side techniques that exploit intra-transaction parallelism [11, 29] divide the transactions into smaller actions and run independent actions in parallel on different nearby cores. Each action has smaller instruction footprint than the entire transaction and a higher chance of fitting its instructions in the L1I cache. On the hardware side, computation spreading through thread migration [4, 9] both uses multiple cores to execute a transaction and makes newer transactions reuse the instructions brought to the L1I cache by the older transactions without any guidance from the software side. A more effective solution, however, would be to involve both software and hardware enhancements to minimize the stall cycles due to instructions.

8. RELATED WORK

There is a large body of related work on workload characterization for database workloads. Barrosso et al. [5] investigated the memory system behavior of OLTP and DSS style workloads using TPC-B and TPC-D [36], respectively, both on a real machine and with a full-system simulation. They found that these two types of workloads need different architectural designs in terms of the memory system. Ranganathan et al. [32] used the same workloads as in [5]. However, they only focused on the effectiveness of out-of-order execution on SMPs while running these workloads in a simulation environment. We believe, neither TPC-B nor TPC-D can be representative of TPC-E since TPC-E has much more complicated and longer-running transactions than TPC-B and it is not completely read-only like TPC-D.

Keeton et al. [22] experimented with TPC-C on a 4-way Pentium Pro SMP machine and performed a similar analysis to [5, 32]. Although, TPC-C is closer to TPC-E compared to both TPC-B and TPC-D, it still has major differences

from TPC-E as described in Section 2. Stets et al. [33] performs a micro-architectural comparison between TPC-B and TPC-C. We add TPC-E to this comparison and also analyze what happens within the storage manager.

Ailamaki et al. [2] examined where the time goes on four different commercial DBMSs with a microbenchmark to have a finer-grain understanding of the memory system behavior of multiprocessors. Hardavellas et al. [15] analyzed OLTP, with TPC-C, and DSS, with TPC-H, on both in-order and out-of-order machines by using a simulation environment. Rather than optimizing the hardware for the workloads, these two papers focused on the implications on the DBMS side in order to utilize the underlying hardware more effectively. In our work, we consider both the hardware and the DBMS design for optimal TPC-E execution.

Johnson et al. [18, 20] and Pandis et al. [29, 30] provide detailed analysis on where the time goes within the storage manager for typical OLTP benchmarks. Their main aim was to highlight components that become scalability bottlenecks in the existing systems and propose alternative designs that remove those bottlenecks. In this paper, we also perform the same analysis with TPC-E and discuss which one of their techniques can or cannot help TPC-E, and also expose the bottleneck on L1 instruction misses.

There are a few performance analysis papers that use TPC-E. For example, [10, 21] use I/O traces of a production database server running TPC-E in order to study its I/O behavior. In [10] the authors compare the I/O behavior of TPC-C and TPC-E. We do not study the I/O behavior. For our experiments we use memory-resident databases and focus on the micro-architectural behavior. Ferdman et al. [13] present a detailed micro-architectural analysis with many types of workloads on Intel X5670 processors, focusing on the architectural design needs of the scale-out workloads. They provide a comparison between the scale-out workloads and server workloads, like TPC-C and TPC-E. In our work, we use a very similar methodology while analyzing the OLTP benchmarks micro-architecturally on our Intel X5660 processors and our high-level conclusions corroborate with their findings. In addition, we perform such a micro-architectural analysis on different hardware platforms to understand the behavior when we switch from an in-order core to an out-of-order one. Moreover, we also demonstrate which components TPC-E stresses within the storage manager as opposed to a pure micro-architectural study. Atta et al. [4] propose computation spreading through thread migration to minimize instruction misses for OLTP workloads. A part of their study also analyzes the instruction and data misses of both TPC-C and TPC-E with a trace simulation study, but not on real hardware. Finally, [23] uses TPC-E to show that a cluster of "wimpy" (low-power Atom-based) nodes is not as energy-efficient as a cluster of traditional server-grade processors (Xeon-based).

9. CONCLUSIONS

In this paper, we present a thorough workload characterization study for TPC-E. We rely on profiling results to determine where the time goes within the storage manager while executing TPC-E on top. Furthermore, we use performance counters to investigate the micro-architectural behavior on two different camps of modern hardware; aggressive out-of-order and lean in-order. We compare TPC-E with previous OLTP benchmarks standardized by TPC, the well-studied TPC-C and the obsolete TPC-B, to better understand what TPC-E-like workloads need both from the software and hardware.

Our study shows that TPC-E observes higher IPC but, at a high-level, has a very similar micro-architectural behavior to its predecessors; it suffers from a high number of L1 instruction cache misses and spends most of its time stalling on memory accesses. Within the storage manager, TPC-E stresses the lock manager the most, like its predecessors, although it gets a higher penalty within the lock manager due to logical lock contention on hot database records.

We believe TPC-E can benefit from the previous design proposals made for OLTP workloads, both from the hardware side and within the storage manager. Running TPC-E on less aggressive processors, with few instruction issues, and processors that have support for SMT increases its IPC value and leads to a better utilization of micro-architectural resources. However, we advocate a more fundamental specialized solution where hardware and software operate together. Such a design can be based on logical partitioning, intra-transaction parallelism, and/or computation spreading to get the best of modern and future hardware for OLTP.

Acknowledgments

We would like to thank all the members of the DIAS and PARSA laboratories at EPFL, Islam Atta, and Duygu Ceylan for their support and feedback throughout this work. We are very grateful to Onur Kocberber and Rene Mueller for sharing their expertise on VTune with us. We also thank the reviewers for their constructive comments and help to improve this paper. This work was partially supported by a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

10. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[3] Anon. et al. A measure of transaction processing power. *Datamation*, 31(7), 1985.

[4] 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.

[5] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *ISCA*, pages 3–14, 1998.

[6] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM TODS*, 8(4):465–483, 1983.

[7] B. M. Cantrill et al. Dtrace. Available at http://dtrace.org.

[8] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *SIGMOD*, pages 383–394, 1994.

[9] 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.

[10] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Record*, 39:5–10, 2010.

[11] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Optimistic intra-transaction parallelism on chip multiprocessors. In *VLDB*, pages 73–84, 2005.

[12] 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.

[13] 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.

[14] G. Graefe, H. Kimura, and H. Kuno. Foster B-trees. *ACM TODS*, 37(3):17:1–17:29, 2012.

[15] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[16] IBM. IBM breaks double digit performance barrier with 10 million transactions per minute, 2010. Available at http://www-03.ibm.com/press/us/en/pressrelease/32328.wss.

[17] Intel. Intel VTune Amplifier XE performance profiler. Available at http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[18] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[19] 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.

[20] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3:681–692, 2010.

[21] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *IISWC*, pages 119–128, 2008.

[22] 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.

[23] W. Lang, J. M. Patel, and S. Shankar. Wimpy node clusters: what about non-wimpy workloads? In *DaMoN*, pages 47–55, 2010.

[24] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.

[25] C. Mohan. ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *VLDB*, pages 392–405, 1990.

[26] 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.

[27] Oracle. cputrack. Available at http://docs.oracle.com/cd/E19683-01/816-0210/6m6nb7m6s/index.html.

[28] Oracle. SPARC supercluster with 27 SPARC T3-4 servers demonstrates world record performance on TPC-C benchmark, 2010. Available at http://www.oracle.com/us/solutions/performance-scalability/t3-4-tpc-c-12210-bmark-190934.html.

[29] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[30] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: page latch-free shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011.

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

[32] 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-VIII*, pages 307–318, 1998.

[33] R. Stets, K. Gharachorloo, and L. Barroso. A detailed comparison of two transaction processing workloads. In *WWC*, pages 37–48, 2002.

[34] 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.

[35] TPC. TPC benchmark B standard specification, revision 2.0, 1994. Available at http://www.tpc.org/tpcb.

[36] TPC. TPC benchmark D standard specification, revision 2.1, 1998. Available at http://www.tpc.org/tpcd.

[37] TPC. TPC benchmark C standard specification, revision 5.11, 2010. Available at http://www.tpc.org/tpcc.

[38] TPC. TPC benchmark E standard specification, revision 1.12.0, 2010. Available at http://www.tpc.org/tpce.

[39] TPC. TPC benchmark H standard specification, revision 2.14.3, 2011. Available at http://www.tpc.org/tpch.