Hassium: Hardware Assisted Database Synchronization

Hillel Avni
Huawei Technologies
hillel.avni@huawei.com

Aharon Avitzur
Huawei Technologies
aharon.avitzur@huawei.com

ABSTRACT

We present Hassium, a hardware assisted database concurrency control, which is optimized for contention. We use HTM (Hardware Transaction Memory) although it is known for overreaction to conflicts. However, Hassium uses HTM to combine the beneficial parallelism of optimistic concurrency control (OCC) with the strength of traditional pessimistic 2PL (Two-Phase Locking). An evaluation on a 22-core HTM-capable multi-core machine shows that Hassium allows for 2X throughput hike on a range of high contention workloads, and guaranties and maintains satisfactory throughput level under increasing number of threads, where the OCC and 2PL methods fail to do that.

1. INTRODUCTION

Multi-core in-memory transactional systems promise significant performance gains over disk-based systems, but recent proposals often fail to deliver consistently high performance, and they suffer from low performance under high contention [18].

This paper presents Hassium, a novel synchronization scheme which synthesizes optimistic and pessimistic concurrency control and HTM, for best performance and stronger progress guarantees under contention. By progress we mean the ability to maintain throughput under increasing number of threads (level of concurrency). We now explain how Hassium works around the HTM transaction size limitation, and then continue to describe the set of features that makes it superior under contention.

1.1 Using HTM

The first obstacle in using HTM, as the synchronization method for in-memory database transactions, is the size limitation. All HTM writes must fit in $L1$ cache size. Due to associativity, this size can be very small and most database transactions will always abort in HTM (we call these aborts capacity aborts).

One direction to deal with this limitation is to cut a big database transaction T to smaller parts and execute each of them with HTM. In [4] they leverage the *transaction chopping* [14] technique for this purpose, but as explained in [21], chopping enforces strict restrictions on T. A more flexible way is to divide T to segments that each accesses a single row (a unit of concurrency control). As T is split to small pieces that execute independently, a higher level synchronization protocol, is required to keep T serializable. In [9] that higher level protocol is a variation of timestamp ordering (TSO). While this approach allowed the creation of a concurrent version of HyPer database [7] with minimal effort, it implies a centralized timestamps generation which is not scalable. In addition, TSO is too restrictive and may not tolerate high contention. In [9] they manage to dynamically partition the workload and minimize contention, to make TSO a scalable higher level protocol, but this is not possible in the general case.

There were suggestions to split an HTM transaction to a read only prefix which is executed without synchronization, followed by validation and writing in HTM [17]. However, this method forces all updates of a transaction to occur within one HTM transaction, which can not accommodate the updates of an arbitrary database transaction. In split hardware transactions from [10] they accumulate read and write sets during small splits of a large transaction and use an HTM transaction to atomically validate the read-set and write the write-set. Again, for database transactions, writing the write-set in a single HTM transaction is not feasible.

Hybrid TM (HyTM) [8] is the name for generic methods to ensure HTM progress, while maintaining parallelism. In HyTM a code segment can run concurrently in HTM, and in pure software if it was not able to commit in HTM. However in HyTM every access to shared memory must be instrumented, and to avoid higher level synchronization, they must include all code, including index access etc. For a database transaction HyTM will always run slowly in software.

In DBX [16] they use use one HTM transaction in the commit phase of an OCC algorithm, and wrap each access with a separate HTM transaction. The main problem with their algorithm is that the commit method in OCC writes, after successful validation, the full write-set to its final destination. In a database transaction the written data will frequently violate the HTM size limitation. In addition, when contention exists in the workload, it is likely that HTM aborts will accumulate fast and hurt the performance.

Hassium approach: Hassium wraps each row access, for read or write, **not** including the index part, within a

	Writer Allows Reader	Writer See Writer	Reader Allows Writer	At Least One Survives
Hassium	**Yes**	**Yes**	**Yes**	**Yes**
OCC	Yes	No	Yes	No
ETL	No	Yes	Yes	No
2PL	No	Yes	No	Yes
HTM	No	No	No	Yes

Figure 1: Concurrency control algorithms and their features

separate HTM transaction, called the *access* transaction. Another HTM transaction is performing the commit, and a separate transaction is used for deadlock detection. However, the HTM transaction that is used for database commit, in Hassium, includes a read-only prefix, followed by a single write, immediately followed by an HTM commit. The same is true for the deadlock detection HTM transaction. The HTM transactions which are used for data access, write the before-image to the undo log. A read never writes shared memory and the write is only writing shared memory once for acquiring the row, followed immediately by an HTM commit. Copying continuous memory is HTM friendly as it does not challenge cache associativity.

Intel's HTM is using large bloom filters to detect conflicts while allowing read-set entries to be evicted from $L1$ cache without aborting the HTM transaction [12]. This way the HTM transaction can accommodate very large read-sets. Thus, Hassium is built on the assumption that a potentially large read-only prefix is tolerable.

1.2 Related Work

Although there have been recent suggestions for efficient in-memory MVCC [11], we chose to avoid the rapid garbage collection, and maintain only actual data. Another design choice of Hassium is not to partition the data, as done in H-Store [5], because in realistic workloads transactions cross partition boundaries and performance rapidly degrades [15]. Some new designs use static and dynamic analysis to regulate parallelism [13], but this approach can introduce high latency and impractical constraints.

Hassium approach: Hassium is a single-version, shared-everything concurrency control method. Below we compare Hassium with competitors from this category.

1.3 Comparable Related Work

Before we go into the details of Hassium, and to an experimental evaluation of it, we make a characteristic comparison to see what makes Hassium unique. We introduce the types of algorithms that share Hassium approach of single-version and shared-everything concurrency control. Then we describe the set of characteristics that make them scalable in contention, and then we show that only Hassium contains all of these characteristics.

We split the Hassium category of algorithms to the following subcategories:

- **Optimistic concurrency control (OCC):** An OCC algorithm, e.g Silo [15] and TicToc [20], has three phases: The transaction reads records from the shared memory and performs all writes to a local, private copy of the records (the *read phase*). Later, the transaction performs a series of checks (the *validation phase*) to ensure consistency. After successful validation, the OCC system commits the transaction by making the changes usable by other transactions (the *write phase*). If the validation fails, the transaction is aborted and nothing is written. If two OCC transaction execute concurrently, they never wait for each other.

- **Encounter time locking (ETL):** In ETL, readers are optimistic, but writers lock the data which they access. As a result, writers from different ETL transactions see each other, and can decide to abort. It was verified empirically in [6] that ETL improve performance of OCC in two ways. First they detect conflicts early and often increase the transaction throughput because transactions do not perform useless work, as conflicts discovered at commit time, in general, cannot be solved without aborting at least one transaction. Second, encounter-time locking allows us to efficiently handle reads-after-writes (RAW) without requiring expensive or complex mechanisms.

- **Pessimistic concurrency control (2PL):** Lock a row at access time for read or for write, and release the lock at commit time. These algorithms require some deadlock avoidance scheme. The deadlock can be detected by calculating cycles in a wait-for graph or avoided by keeping time ordering in TSO [2] or by some back-off scheme. In 2PL algorithms, if one transaction is writing a row, no other transaction can access it, and if a row is being read, no transaction is allowed to write it.

- **Hardware transactional memory (HTM):** Current implementations of HTM do not allow any access concurrent with a write, but if two transactions conflict, one of them survives, according to the *requester-wins* policy [1].

After we saw the methods comparable with in Hassium, we can stipulate what characteristics of each algorithm make it scalable under contention:

- **Writer Allows Reader:** If two live transactions T_1 and T_2 access a row R, Hassium allows T_1 to read R after T_2 wrote R. This is good because there is a chance that both T_1 and T_2 will later commit successfully, if T_1 will commit before T_2. Hassium shares this feature with OCC.

- **Reader Allows Writer:** If two live transactions T_1 and T_2 accesses a row R, Hassium allows T_2 to write R after T_1 read R. Again, there is a chance that both T_1 and T_2 will later commit successfully, if T_1 will commit before T_2. Hassium shares this feature with OCC and ETL.

- **Writer see Writer:** If T_1 access R for write, and T_2 also needs to write R, T_2 will see T_1 is writing R, and

may abort or wait as long as it does not introduce a deadlock. OCC allows both T_1 and T_2 to proceed after writing R but this implies wasted work as one of them will later have to abort. Hassium shares this feature with 2PL and ETL.

- **At Least One Survives:** In the case that T_1 wrote R_1 and T_2 wrote R_2, and then T_1 read R_2 and T_2 read R_1, in Hassium, either T_1 or T_2 will always survive. This is not true for OCC, where in commit T_1 would lock R_1, T_2 would lock R_2, and in validation, if this occurs simultaneously due to contention, both will see a locked item in their read set and will abort. Hassium shares this feature with 2PL and HTM.

To summarize we show in Figure 1 the set of method subcategories and the set of characteristics which they share with Hassium. As shown, none of the existing algorithms accommodates all the features which make Hassium scalable under contention.

The remainder of this paper is organized as follows. In Section 2 we present the Hassium algorithms and in Section 3 we prove its correctness and in Section 4 performance results of Hassium are reported. We conclude in Section 5.

2. HASSIUM

Hassium uses small HTM transactions which read or acquire a single row, and use another transaction to commit the database transaction. Unlike [9] the readers do not write anything into the rows, but unlike OCC, the writers are pessimistic. Hassium breaks the database transactions to code segments (splits) that each accesses a single row, and exploits HTM to execute efficiently, both the splits and their reassembly. The following HTM transactions can participate in a Hassium database transaction:

- **Read:** Locates the last committed version of the row and reads the data.
- **Write:** This transaction simultaneously:
  - Copies the row to the undo-set.
  - Links the row to the committed version which was copied to the undo-set.
  - Locks the row.

  The data is written in place, out of the HTM context, and concurrent read HTM transactions use the committed data which is linked from the row to the undo-set.
- **Commit:** Validates the read-set and commits the database transaction.
- **Deadlock prevention:** If a write sees a concurrent write, it can just abort immediately, or call this transaction to prevent deadlocks and wait.

The Hassium HTM transactions fit within the HTM size limitations so capacity violations practically do not exist. The HTM aborts are due to conflicts on the same cache line, i.e., multiple HTM transactions which access the same cache line and at least one of them is writing. An important part of Hassium is to reduce these conflicts.

Hassium is designed to be lightweight and minimize both HTM and database aborts, even when contention is high. The low abort rate and high throughput of Hassium in workloads with high contention represents a new exploration of HTM and is one of Hassium contributions. Still, in extremely high contention, HTM conflicts problem exists.

Database-related HTM aborts are triggered by the software when it sees a potential conflict at the database level. These aborts are named explicit aborts in HTM terminology. The need for explicit aborts forced us to use the restricted transactional memory (RTM) mode of Intel TSX feature [1] instead of the hardware lock elision (HLE) mode. HLE is backward compatible with traditional locking and used in [9]. However, in HLE abort, the hardware automatically takes the lock, while in RTM, software is able to trigger and handle aborts. In Hassium, we take advantage of the RTM flexibility, and abandon the HLE backward compatibility.

2.1 Algorithms

Hassium is executed by a set of concurrent worker threads. Each worker thread has a unique ID (tid) and a local monotonous increasing version counter (tv). In addition, a global Last Committed versions Array (lca) is maintained. A database transaction is uniquely identified by its tid and tv.

Each thread has a slot in the lca and upon a successful database commit, it writes tv in its slot in the lca, i.e., $lca[tid] \leftarrow tv$, and then increments tv locally. A database row also has the attributes rid and rv, which are the tid and tv of the last transaction, T, that wrote it. If T is live, i.e., uncommitted, the row has a link to $prev$ which is the last committed version of the row, including its rid, rv and data. The $prev$ link is valid only while a live transaction is writing the row. In total, the $prev$ links point to the undo-set of the live database transaction T.

We split the database transaction to small HTM transactions that access a row for read or for write, and use another HTM transaction to perform validation and commit of the database transaction. If a writer sees a row is being written by another live database transaction, it checks for deadlocks to decide if to wait or abort the database transaction.

The commit HTM transaction may be larger, but it is read-only until it finally writes to the lca and immediately commits.

2.2 Starting HTM and Handling Conflicts

In this section we see how an HTM transaction is triggered and conflicts are handled in Hassium *Access* function. The *Access* function starts with a call to the function $HsBeginAccess$ that is shown in 1. It calls $\_xbegin$ which either returns success or the type of abort. In case of an explicit abort, the $\_xbegin$ also returns a user code for the abort reason.

We discuss the `LOCKED` abort code processing in Section 2.3.2, and here we look into the `FALLBACK` abort code processing. If the HTM transaction got more than threshold of conflicts, it must assume the HTM can not commit successfully and fallback to software mode. The simplest fallback mode is a global lock [1], but this solution serializes the system.

In Hassium, in case of fallback, we only lock the row. As the *Access* accesses a single row, it will never deadlock, and the lock is released in the $HsEndAccess$ function. We discuss the delimiters of $ValidateCommit$ and $HsDeadlock$

Algorithm 1 Starting HTM

```
1: function HsBeginAccess(row)
2:     while true do
3:         status = _xbegin
4:         if status = _XBEGIN_STARTED then
                                   ▷ HTM Started successfully
5:             if row.is_locked() then
6:                 _xabort(FALLBACK)
7:             end if
8:             return true
9:         end if
10:        if status = _XABORT_EXPLICIT then
11:            if abort_code = LOCKED then
12:                if HsDeadlock(row.rid ⊕ row.rv) then
                                           ▷ Prevent deadlock
13:                    return false
14:                end if
15:                if abort_code = FALLBACK then
                               ▷ Concurrent fallback in progress
16:                    continue
17:                end if
18:            end if
19:        else
20:            Increment(aborts_count)
21:        end if
22:        if aborts_count > threshold then
23:            row.lock()
24:        end if
25:    end while
26: end function
```

and how they maintain consistency and progress in Sections 2.4 and 2.5. If the *Access* function will see the row is acquired by a concurrent transaction in fallback path, it triggers an explicit HTM abort with `FALLBACK` code.

2.3 Access a Row

The pseudo code for the *Access* function is in Algorithm 2. It starts an HTM transaction in line 2, and the whole function is executed in transactional context. If the *HsBeginAccess* returned *false*, then this is a write access which is in a deadlock, and returns `NULL` in line 4 to abort the database transaction.

In line 7 the *Access* checks if the row was already written by the executing database transaction, by comparing the *rid* and *rv* to their current local values, i.e., *tv* and *tid*. If it is an access after write, the row is reused for the current access, and nothing is recorded in the read and undo sets. At this point we see the low overhead of the pessimistic writes.

If this is not an access after write, In line 12, if the version of the row (*rv*, *rid*) is committed according to the *lca*, *committed* is set to *true*. Otherwise, the row is currently being written by a live transaction, and committed is set to *false*. At this point the function splits to different write and read paths.

2.3.1 Read

In line 16 If the row is committed, *lc*, i.e. pointer to Last Committed version of the row, is set to the row itself. Otherwise, the *lc* is set in line 19 to *row.prev*. In the next section we show how the write sets the *row.prev* field. In

Algorithm 2 Getting access to a row

```
1: function Access(row, type)
2:     if HsBeginAccess(row) = false then
3:
4:         rc ← NULL
5:         goto 35
                                   ▷ Deadlock, so abort
6:     end if
                                   ▷ Started HTM
7:     if row.rid = tid ∧ row.rv = tv then
8:
9:         rc ← row
10:        goto 34
                ▷ Access after write by same transaction
11:    end if
12:    if lca[row.rid] ≤ row.rv then
13:        committed ← true
                                   ▷ Row is committed
14:    end if
15:    if type = read then
16:        if committed then
17:            lc ← row
                                   ▷ Row is committed
18:        else
19:            lc ← row.prev
                ▷ Row is written by a live transaction
20:        end if
21:        rs ← (row, lc.rid, lc.rv)
                        ▷ Add to read-set for validation
22:        rc ← copy(lc)
                ▷ Copy last committed version for user
23:        goto 34
24:    end if
                                   ▷ It is a write access
25:    if committed then
26:        e ← copy(row)
                            ▷ Create an undo-set entry
27:        row.prev ← e
28:        row.rid ← tid
29:        row.rv ← tv
30:        rc ← row
                                   ▷ Row is locked
31:    else                ▷ Live writer, so abort HTM
32:        _xabort(LOCKED)
33:    end if
34:    HsEndAccess(row)
                        ▷ Successful write, Commit HTM
35:    return rc
36: end function
```

line 21 the *rv* and *rid* of *lc* are recorded in the read-set (*rs*), together with a pointer to the row itself.

2.3.2 Write

In line 25, as previously done in line 16 for the read access, *committed* is checked, and if there is no concurrent writer who acquired the row, an undo-set entry is created in line 26 and linked to the *prev* pointer in line 27. However, if there is a live concurrent transaction that writes the row, an HTM abort is triggered in line 32 with the code `LOCKED`. The

execution jumps to the HTM abort path in $HsBeginAccess$ function (from Algorithm 1), and in line 11 it sees the `LOCKED` code, and in line 12 it calls $HsDeadlock$ to see if it can wait, i.e. retry the access, or abort to prevent a deadlock. In case there is a deadlock, $HsBeginAccess$ will return $false$ and not restart the HTM. In response, in line 4 the *Access* function will set the output row to `NULL`, to trigger a database abort, and exit the function.

Algorithm 3 Validate and commit or abort

```
1: function ValidateCommit(T)
2:     status ← commit
3:     HsBeginV
                                              ▷ Start HTM
4:     for e ∈ T.rs do
5:         if e(rid, rv) = e.row(rid, rv) then
6:             continue
7:         end if
8:         if lca[e.row.rid] ≥ e.row.rv then
                                  ▷ Newer writer is committed
9:             status ← aborted
10:            break
11:        end if
                          ▷ Current writer (possibly self) is live
12:        if e.row.prev(rid, rv) ≠ e(id, rv) then
                            ▷ Already different committed data
13:            status ← aborted
14:            break
15:        end if
16:    end for
17:    if status = commit then
18:        lca[tid] = tv
19:    end if
20:    HsEndV
                                              ▷ Commit HTM
21:    if T.us ≠ ∅ then
         ▷ Writing transaction: undo-set (us) is not empty
22:        lfa[tid] = tv
23:        increment(tv)
                                ▷ Increment, cannot reuse the tv
24:    end if
25:    if status = aborted then
26:        rollback(T)
27:    end if
28:    cleanup(T)
                                ▷ Truncate read-set and undo-set
29: end function
```

2.4 Validation and Commit

The pseudo code for the validation and commit function is in Algorithm 3. From line 3 to line 20 it executes an HTM transaction to verify that the values accessed by the optimistic readers are not overwritten by newer committed database transactions, and if the validation passes it commits the values written by the pessimistic writers, by setting the transaction version (tv) in the lca in line 18. After the HTM transaction, if the transaction was a writing transaction, i.e. in line 21 the undo-set is not empty, the code sets the current tv in the lfa, i.e. the Last Finished transactions Array, and then increments the local tv in line 23. We can not reuse tv for writing, as it is already marked as finished in the lfa, and can lead to deadlocks. The lfa is used **only in deadlock checking**, were it does not matter if the database transaction committed or aborted, only if it is finished, or live and may hold locks. If status is *aborted*, a rollback is executed in line 26.

To verify the readers saw a consistent view, i.e., the last committed values, we have a two-step validation. If the rv and rid logged by the read for this row are unchanged, as verified in line 5, the reader saw the last committed values and can continue in line 6. The second case where a read can be valid is when the current rv and rid of the row are different than the logged ones, but they represent a live transaction and the logged rv and rid are linked from the $prev$ of the row, so the logged data is still the last committed version. In line 8 the the current rv and rid of the row are checked to see if they represent a committed transaction, and if they do, abort in line 9. In line 12 the logged rv and rid are compared to the last committed ones in $row.prev$, and if they are equal, then the version logged by the transaction is the last committed one. Otherwise, the transaction aborts in line 13.

In case of excessive HTM aborts, the $HsBeginV$ is taking a fallback lock which serializes all validations. In case a validation executes in fallback mode, It must lock the read set, and when it is HTM mode it must check the row is not locked by *Access* fallback. This can not create a deadlock, as there is only one $ValidateCommit$ in fallback, and each *Access* execution can hold up to one row. However, the $HsBeginV$ is counting rows that are locked for a fallback as conflicts towards a fallback of the $ValidateCommit$ function. In case of a fallback of $ValidateCommit$, it must release the fallback locks from all the read-set before calling $HsEndV$ to release $ValidateCommit$ fallback lock.

Rollback: If the transaction aborts, it rolls back in non-transactional context. The data is restored, and then the last committed rv and rid are written by a single store instruction per row. only after the committed version is restored, the row actual data becomes visible to concurrent accesses.

2.5 Deadlock Detection

We exploit HTM and the tid, tv, rid and rv we have in Hassium to devise a very efficient deadlock detection mechanism, which is presented in Algorithm 4. The wait-for graph (WFG) is simply an array where each thread has a slot at the index which corresponds to its own tid. The *blocker* is the rid and rv of the row the transaction needs to write. The function starts an HTM transaction and then traverses the WFG, where in line 11 it checks if, according to the last **finished** array (lfa) the thread whose tid is the slot index is waiting for a live transaction. If yes, in line 19 it goes to the slot whose index is the rid written in this slot, starting from the slot index $blocker.rid$. If there is no deadlock, i.e. the traversal reached a committed/aborted blocker, according to the lfa in line 11 the function writes the original blocker in its slot in line 28 and immediately commits the HTM. When a deadlock occurs, there is exactly one thread that identifies it. As seen in lines 16 and 17, it chooses the transaction with the smallest write-set, a.k.a. karma, and if there was a deadlock, terminates that transaction in line 28. In all cases, if a transaction was the victim, it returns true, to signal there was a deadlock, and if the blocker transaction commits, it returns false to resume caller transaction.

Fallbacks: In case $HsBeginDL$, in line 7, is encountering

Algorithm 4 Deadlock Detection and Prevention

```
1: function HsDeadlock(blocker)
2:     lock_holder ← blocker
                              ▷ Keep the original lock holder
3:     deadlock ← false
4:     victim ← tid
5:     karma[tid] ← write_set_size
6:     min_karma ← karma[tid]
7:     if HsBeginDL = false then
                                                  ▷ Start HTM
8:         return true
9:     end if
10:    while deadlock = false do
11:        if lfa[blocker.tid] ≥ blocker.tv then
                                          ▷ blocker is committed
12:            WFG[tid] ← lock_holder
13:            break
14:        else
                                          ▷ Update the victim
15:            if karma[blocker.tid] < min_karma then
16:                min_karma ← karma[blocker.tid]
17:                victim ← blocker.tid
18:            end if
                   ▷ Go to the next transaction in the WFG
19:            blocker ← WFG[blocker.tid]
20:        end if
21:        if blocker.tid = self.tid then
                   ▷ If the blocker is self, we have a deadlock
22:            deadlock ← true
23:        end if
24:    end while
25:    HsEndDL
                                                  ▷ Commit HTM
26:    if deadlock = true then
27:        WFG[tid] ← lock_holder
28:        WFG[victim] ← ⊥
29:    end if
30:    while true do
31:        if lfa[lock_holder.tid] ≥ lock_holder.tv then
                                                  ▷ No deadlock
32:            return false
33:        end if
34:        if WFG[tid] = ⊥ then
                                          ▷ There was a deadlock
35:            return true
36:        end if
37:    end while
38: end function
```

too many aborts, it returns *true*, and the database transaction is aborted. In our experiments, this scenario is very rare. The deadlocks detection never takes locks.

3. CORRECTNESS

Before we continue to the details of Hassium interaction with HTM, we make sure it is correct, that is, serializability is maintained and progress is guaranteed.

3.1 Safety

A transaction T_i is a set of reads $r_i(x)$ and writes $w_i(y)$ followed by a commit operation c_i. According to [3] two operations are said to *conflict* if they both operate on the same data item and at least one of them is a write. We say that an operation $o_i(x)$ precedes in a conflict an operation $o_j(x)$ if $o_j(x)$ is a read and $o_i(x)$ is a write, and $o_j(x)$ read what $o_i(x)$ wrote, or if both $o_i(x)$ and $o_j(x)$ are writes and the final value of x is written by $o_j(x)$.

The serialization graph of an execution, is a directed graph whose nodes are the committed transactions and whose edges are all $T_i \rightarrow T_j$, $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations. The serializability theorem [3] maintains that an execution is serializable if it creates an acyclic *serialization graph*.

Lemma 1. *Hassium forces strict order among the HTM transactions c_i and c_j that represent the commit of the corresponding database transactions T_i and T_j.*

Proof. As both c_i and c_j are executed in HTM transactions, i.e. lines 3 to line 25 in Algorithm 3, and as HTM has single global lock semantics, either $c_i \rightarrow c_j$ or $c_j \rightarrow c_i$. □

Therefore in Hassium executions, transactions have inherent order, and we say T_i precedes T_j if c_i precedes c_j.

Lemma 2. *If two operations $o_i(x)$ and $o_j(x)$ conflict, if $o_i(x)$ precedes $o_j(x)$ than T_i precedes T_j.*

Proof. We assume by contradiction that $o_j(x)$ precedes $o_i(x)$ in a conflict but T_i precedes T_j. If $o_i(x)$ read from or wrote on the same row (item x) that $o_j(x)$ wrote, than in line 12 of Algorithm 2 T_i saw the tv of T_j in lca, which means T_j committed in line 18 of Algorithm 3 while T_i was still alive. □

From lemmas 1 and 2 we conclude that if there is an edge from T_i to T_j in a serialization graph, than $i < j$ and therefore the graph is acyclic, and according to the serializability theorem:

Theorem 3. *Transactions that follow Hassium algorithm are serializable.*

3.2 Liveness

First, we note that our deadlock detection scheme always lets the most advanced transaction continue, so deadlocks will not happen. Second, we show that a livelock is impossible. Namely, if T_i writes on a row that T_j reads, and T_j writes on a row T_i reads, they might both abort and go into infinite retries. However, this is not possible, as T_i will cause T_j to abort in line 8 of Algorithm 3, only if T_i committed, and then progress has been made.

4. EXPERIMENTAL EVALUATION

We now present our evaluation of Hassium. For these experiments, we use the DBx1000 OLTP DBMS [19]. This is a multi-threaded, shared-everything system that stores all data in DRAM in a row-oriented manner with hash table indexes. DBx1000 uses worker threads (one per core) that invoke transactions from a fixed-length queue. Each transaction contains program logic intermixed with query invocations. Queries are executed serially by the transaction worker thread as they are encountered in the program

logic. Transaction statistics, such as throughput and abort rates, are collected after the system achieves a steady state during the warm-up period. DBx1000 includes a pluggable lock manager that supports different concurrency control schemes.

4.1 Hardware Settings

We deployed DBx1000 on a 22-core machine with four Intel Xeon E5-2699-v4 CPUs and 250 GB of DRAM. Each core supports two hardware threads, but we disabled them as hyperthreading can make HTM non deterministic.

4.2 Comparison With Other Algorithms

To see the contribution of the different characteristics to performance, we use three variants of Hassium:

1. **HS:** The Hassium version from Section 2.
2. **HS-NDL:** Hassium with No DeadLock - To check the contribution of deadlock detection, we use Hassium where a writer that sees an uncommitted data, i.e. a row acquired by a concurrent writer, immediately aborts.
3. **HS-ETL:** To check the contribution of a writer allows reader, we use a Hassium flavor where a reader or a writer that sees an uncommitted data, immediately aborts.

We compared Hassium to the following algorithms:

1. **TSO-HTM:** This is our implementation of [9].
2. **2PL-DL:** The DL-DETECT algorithm from [15]. A pessimistic two phase locking algorithm with deadlock detection.
3. **2PL:** The NO-WAIT algorithm from [15]. The pessimistic two phase locking without deadlock detection, which always aborts and backoff if it fails to lock.
4. **TICTOC:** This is the original implementation from [20]. The most scalable to contention OCC algorithm we are aware of.
5. **DBX-HTM:** The HTM based OCC algorithm from [16]. HTM is used to avoid locking in commit and data access.

We do not show the plain Intel HTM [1] applied to entire database transactions, as we saw it can not scale to arbitrary transactions or contention.

4.3 Workloads

We next describe the three benchmarks that we executed in the DBx1000 testbed [20] for this analysis. First we show the *impossibility test* which shows Hassium robust throughput under high concurrency, second is TPC-C and third is YCSB benchmark.

4.3.1 Impossibility Test

We created a synthetic, YCSB like workload. There is a table with n rows ($r_1 ... r_n$) and n connections threads ($t_1 ... t_n$). A transaction on thread t_k writes r_k, and then reads all the other rows and commit. While this is not a realistic workload, it represents scenarios that can occur in the life of a database. When we executed this workload with the different concurrency controls, all except HS and HS-NDL cannot perform under high concurrency and did not progress at all. We expected this behavior from the OCC TICTOC algorithm. However, the 2PL-DL also encountered repeatable deadlocks and HS-ETL always aborted as a reader saw a locked write.

While it is possible to guarantee progress in OCC by serializing commits, or in 2PL by grabbing a global lock, these techniques can have bad effect on the whole system performance. Hassium manages to progress as is, due to its characteristics, i.e. writers do not stop readers and the *At Least One Survives* characteristic we describe in Section 1.3.

Figure 2 shows that both HS and HS-NDL manage to maintain progress in this unscalable workload. All other concurrency controls could not progress already with two cores. We did not check Cicada [11], as it is MVCC, but to our understanding, a Cicada validation will see `PENDING` versions in all its read-set and block or backoff, i.e. will not make progress either.

4.3.2 TPC-C

This workload is the current industry standard to evaluate OLTP systems. It consists of nine tables that simulate a warehouse-centric order processing application. Only two (Payment and NewOrder) out of the five transactions in TPC-C are modeled in our simulation. These two make up 88% of the default TPC-C mix and are the most interesting in terms of complexity for our evaluation.

4.3.3 YCSB

The Yahoo! Cloud Serving Benchmark is representative of large-scale on-line services. Each query accesses a single random tuple based on a Zipfian distribution with a parameter (theta) that controls the contention level in the benchmark.

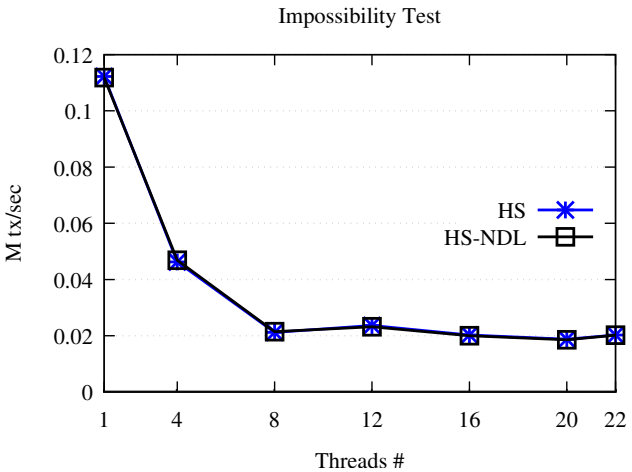
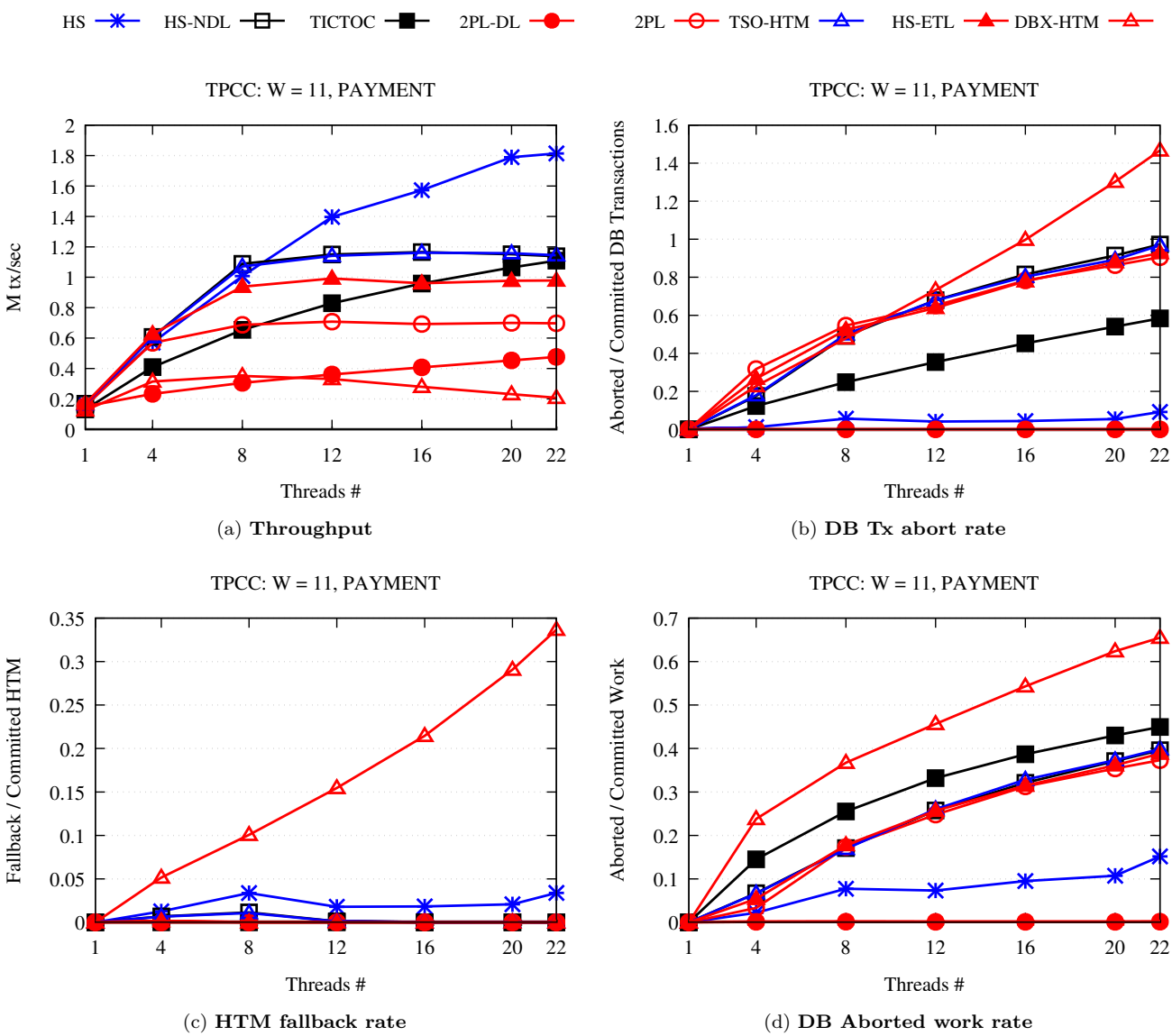


Figure 2: Throughput for the impossibility test.

4.4 Performance data

For each workload we created four graphs, besides throughput, to understand the root of the performance. Each point in the graphs is the average of 6 executions:



(a) **Throughput**

(b) **DB Tx abort rate**

(c) **HTM fallback rate**

(d) **DB Aborted work rate**

Figure 3: TPC-C Payment transactions with eleven warehouses, throughput and various abort rates graphs

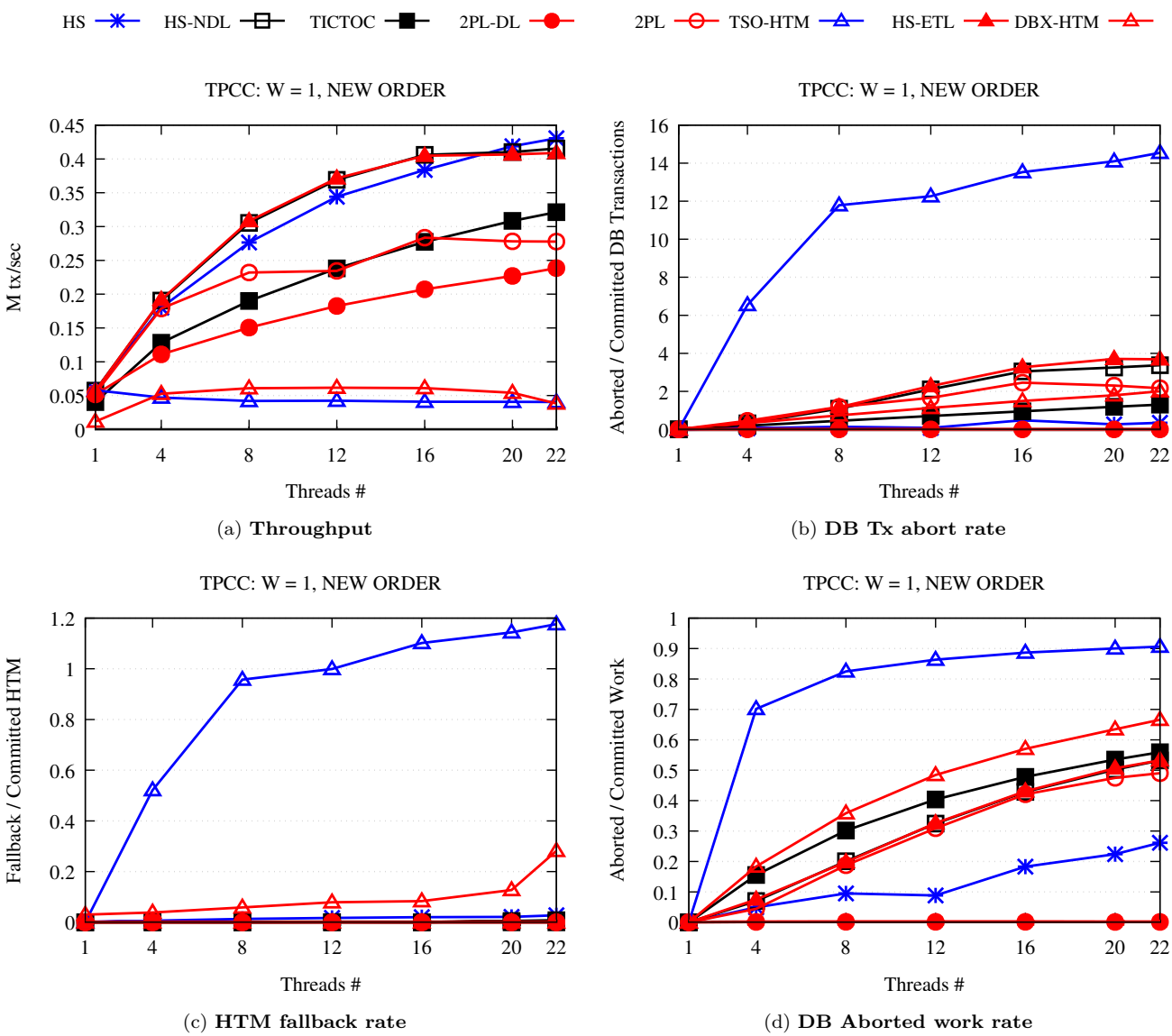
1. **Throughput:** Performance measured in transactions per second.
2. **Aborted database transactions:** The part of the database transactions that were aborted, out of the committed number of database transactions.
3. **Aborted work:** Part of the time spent in executing transactions that were eventually aborted, out of the total execution time.
4. **Aborted HTM:** Part of HTM transactions that were aborted due to conflicts, out of total HTM transactions that were started. Capacity aborts are almost extinct, and we do not show explicit user aborts, as they are not counted towards going to the fallback path.
5. **Fallback HTM:** Part of HTM transactions that were aborted more than 20 times due to conflicts, and had to take the software path.

4.5 Performance analysis

This section presents results for TPC-C and YCSB, which allows us to see how Hassium performs in a range of contention levels.

4.5.1 TPC-C Results

We show Payment with eleven warehouse in Figure 3, so contention gets high with more than 11 cores. The first insight from the throughput graph of payment (Figure 4a), is that with high contention deadlock detection improves Hassium performance significantly. This is the only difference between HS-NDL and HS, and in high contention (22 cores) HS is seventy percent faster than HS-NDL. The reason is that there is a lot of contention but no deadlocks so work can be saved. Note that OCC cannot implement deadlock detection, so this is an important advantage of Hassium over OCC.



(a) **Throughput**

(b) **DB Tx abort rate**

(c) **HTM fallback rate**

(d) **DB Aborted work rate**

Figure 4: TPC-C NewOrder transactions with one warehouse, throughput and various abort rates graphs

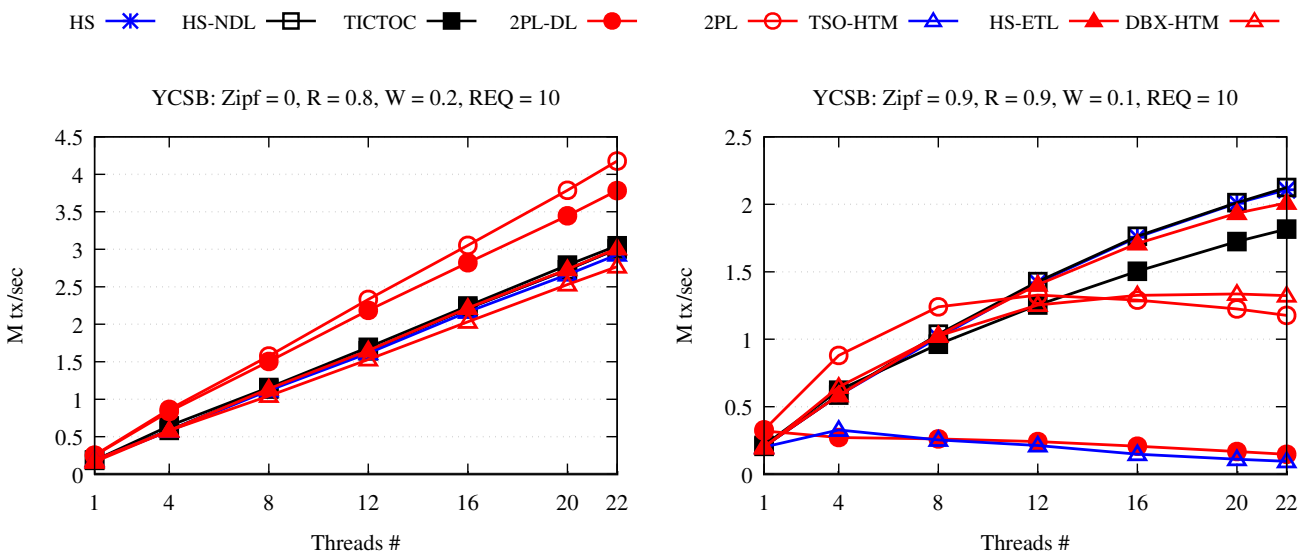
Another phenomenon we see is that the HS-NDL and HS-ETL have higher abort count in Figure 3b than TICTOC, but TICTOC has higher amount of aborted work as seen in Figure 3d. The reason is that both HS-NDL and HS-ETL have early aborts, while TICTOC is aborting only at the completion of the transaction, so each of its aborts is losing more work. Again, this is an inherent advantage of Hassium over OCC which can not implement early aborts.

Compared to 2PL, HS has less aborts, and 2PL-DL is simply having higher overhead for deadlock detection, as it has the lowest DB abort rate, but also the lowest performance.

The DBX-HTM which is doing the commit and the access in HTM, has the lowest performance under contention. The reason can be seen in the HTM fallback rate. We verified that practically all the HTM aborts happen in successful DB commits at the write-back phase. If we extract the writing back from HTM, DBX-HTM performs even better than TICTOC. However, as most successful commits fail to commit in HTM, other transactions proceed, and the amount of database aborts in DBX-HTM grows. Together, the HTM aborts and the database aborts are making DBX-HTM the slowest algorithm for TPC-C payment transaction.

We show NewOrder with one warehouse in Figure 4. There are 10 districts so contention gets high with more than 10 cores. In NewOrder the deadlock detection is not helping and actually is has some overhead as seen from the fact that HS-NDL is somewhat faster than HS. The root of HS variants better performance is again, less aborted work due to early aborts compared to OCC, and reads which progress concurrently with writes, unlike 2PL.

The worst performance in NewOrder belongs to TSO-HTM [9] as it writes in read accesses which increases its HTM aborts and fallbacks as seen in Figure 4c, as well as



(a) Low contention (zipf = 0) YCSB workload

(b) Low - medium contention (zipf = 0.9, 10% writes) YCSB workload

Figure 5: Throughput graphs for YCSB workloads with Low contention.

aborted transactions rate 4b and aborted work 4d.

4.5.2 YCSB Results

We continue to YCSB where it is easier to control the workload characteristics.

4.5.2.1 Low Contention.

To summarize, we see that Hassium is optimized for high contention, and there it is the best. For low contention, as seen in Figure 5a the 2PL are better. As in low contention no serialization is caused by locking, the overhead of the OCC, e.g. versioning, validation and bookkeeping, makes it less optimal.

Figure 5b shows a workload with zipf 0.9 but only 10% writes. Up to 8 cores TICTOC and Hassium are equal but then Hassium lower number of aborts lets it pass TICTOC. At 12 cores, 2PL stops scaling and Hassium passes it too. Thus, this workload on 12 cores marks a contention where Hassium becomes the most efficient concurrency control.

4.5.2.2 High Contention.

There are three parameters that determine the contention level and contention type in a YCSB workload: the theta of the zipfian distribution, the portion of writes out of the total access, and the number of requests per transaction.

With 10 requests, When theta= 0.8 and 50% of the access are writes, in Figure 6a, we can see HS is best from 12 cores were it passes 2PL which stops scaling. However from 20 to 22 cores HS performance drops and the reason is shown in Figure 6b which shows the HTM abort rate in 22 cores is 1, which means on average each HTM transaction experienced an abort. The HS-NDL has fewer aborts and scales better from 16 cores. The reason is that in very high contention the deadlock-detection does introduce some HTM contention. The HS-ETL performance is as HS but it shows few aborts. This is misleading as in HS-ETL a transaction gets aborted when it sees a locked row, but these are explicit aborts and are not counted by the framework.

Figure 6c shows the same workload as Figure 6a but with theta = 0.9. This change increases the HS abort rate, as seen in Figure 6d so now it gets to 1 already in 12 cores, and in 22 the abort rate of HS is 3, i.e. each HTM transaction is retried 4 times on average. From 16 cores, when HTM abort rate reaches 2, the HS performance drops below TICTOC. The reason for the high HS abort rate is that each transaction sees writes from multiple other transactions and then they conflict in the database commit HTM transaction on the *lca* slots.

In Figure 6e the theta is 0.95, i.e. the zipfian distribution is much more contentious. However, we reduced the number of requests to 5, so each live transaction sees less live transactions on average. As a result the HTM abort rate of HS reaches 1 only at 22 cores and it stays on top for the whole experiment. On 22 cores HS and TICTOC are equal but the HS-NDL and HS-ETL variants still win.

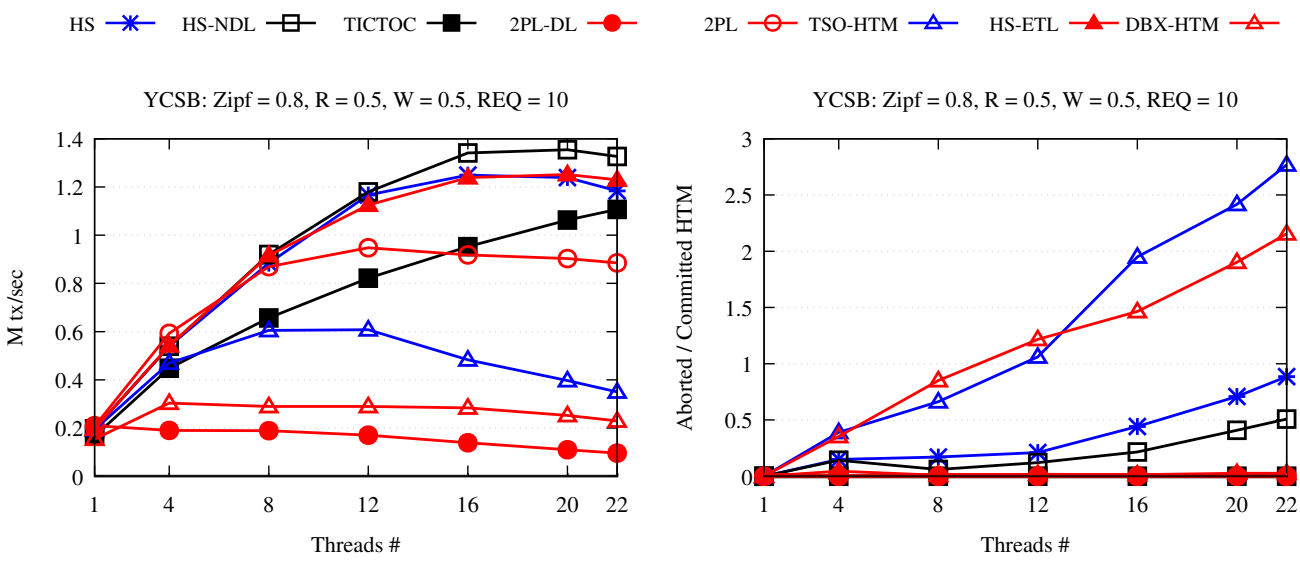
5. CONCLUSION AND FUTURE WORK

We have presented Hassium, a novel concurrency control which uses HTM, as well as principles from optimistic and pessimistic approaches. We show in experiments that for a range of high contention database workloads, Hassium has up to 2X better performance than state of the art synchronization algorithms that use, or do not use HTM.

In addition we showed both theoretically and in experiment that Hassium, although it uses HTM, has stronger progress guarantees than software based 2PL and OCC. In fact, we show a workload where Hassium manages to commit transactions with 22 threads, while all other concurrency control algorithms throughput drops to zero with two threads.

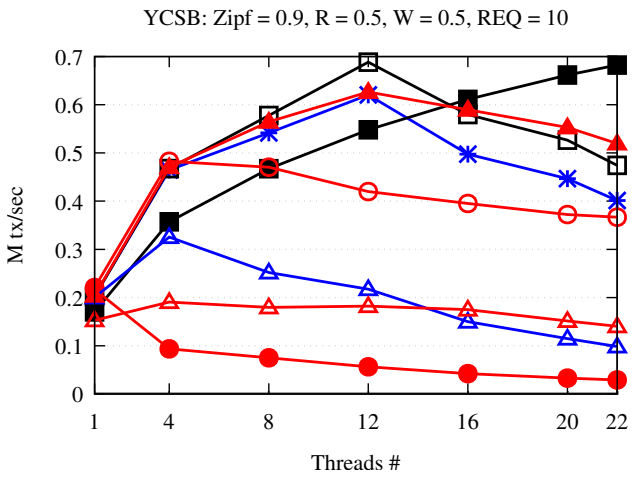
6. REFERENCES

[1] Intel architecture instruction set extensions programming reference.

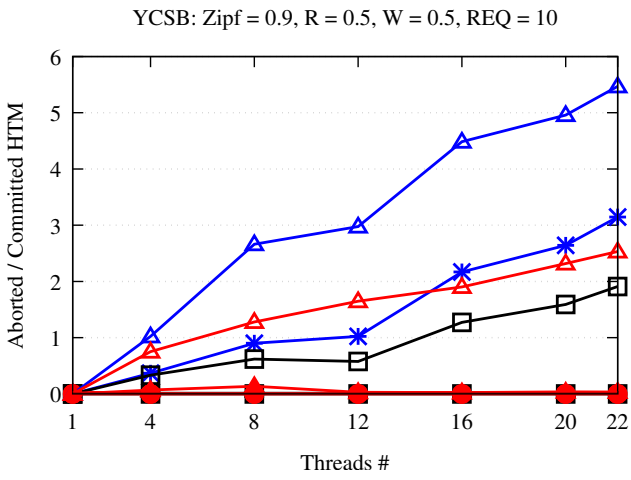


(a) High contention - **Throughput**

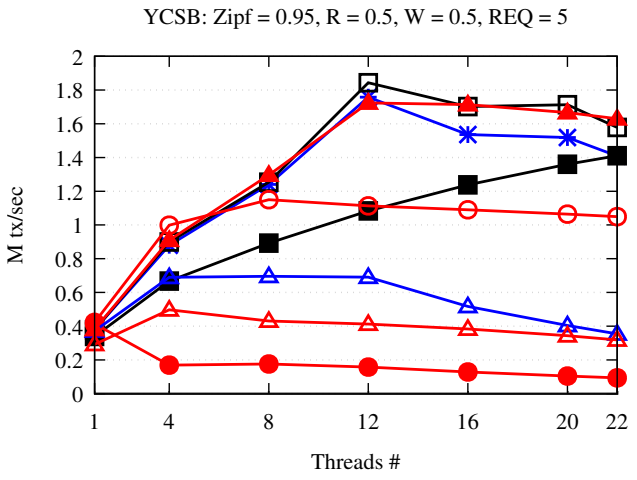
(b) High contention - **HTM Aborts**



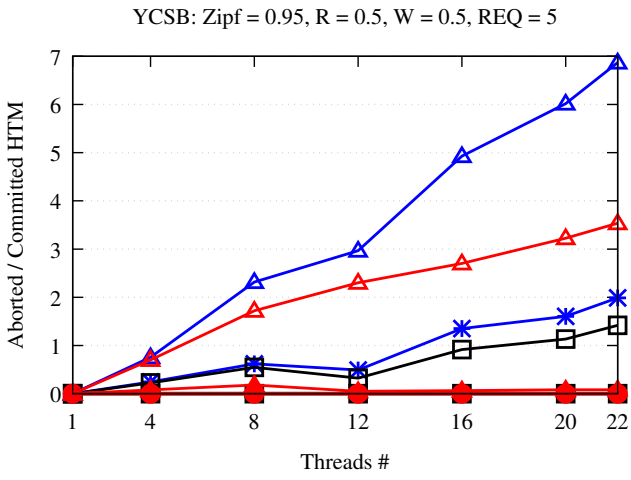
(c) Very High contention - Long Transaction - **Throughput**



(d) Very High contention - Long Transaction - **HTM Aborts**



(e) Extreme contention - Shorter Transaction - **Throughput**



(f) Extreme contention - Shorter Transaction - **HTM Aborts**

Figure 6: YCSB with high and extreme contention. Each workload has throughput on the left and HTM abort rate on the right.

[2] Bernstein, P. A., and Goodman, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (1981), 185–221.

[3] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] Chen, Y., Wei, X., Shi, J., Chen, R., and Chen, H. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the 12th European Conf. on Comp. Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016* (2016), pp. 26:1–26:17.

[5] Faerber, F., Kemper, A., Larson, P., Levandoski, J. J., Neumann, T., and Pavlo, A. Main memory database systems. *Foundations and Trends in Databases 8*, 1-2 (2017), 1–130.

[6] Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008* (2008), pp. 237–246.

[7] Kemper, A., Neumann, T., Funke, F., Leis, V., and Mühe, H. Hyper: Adapting columnar main-memory data management for transactional AND query processing. *IEEE Data Eng. Bull. 35*, 1 (2012), 46–51.

[8] Kumar, S., Chu, M., Hughes, C. J., Kundu, P., and Nguyen, A. D. Hybrid transactional memory. In *Proc. of the ACM SIGPLAN Sym. on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006* (2006), pp. 209–220.

[9] Leis, V., Kemper, A., and Neumann, T. Scaling htm-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng. 28*, 2 (2016), 297–310.

[10] Lev, Y., and Maessen, J.-W. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 197–206.

[11] Lim, H., Kaminsky, M., and Andersen, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 21–35.

[12] Matveev, A., and Shavit, N. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015* (2015), pp. 59–71.

[13] Ren, K., Faleiro, J. M., and Abadi, D. J. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1583–1598.

[14] Shasha, D., Llirbat, F., Simon, E., and Valduriez, P. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst. 20*, 3 (Sept. 1995), 325–363.

[15] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.

[16] Wang, Z., Qian, H., Li, J., and Chen, H. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 26:1–26:15.

[17] Xiang, L., and Scott, M. L. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015* (2015), pp. 76–86.

[18] Yu, X., Bezerra, G., Pavlo, A., Devadas, S., and Stonebraker, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB 8*, 3 (2014), 209–220.

[19] Yu, X., Bezerra, G., Pavlo, A., Devadas, S., and Stonebraker, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 209–220.

[20] Yu, X., Pavlo, A., Sanchez, D., and Devadas, S. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 1629–1642.

[21] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., and Li, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), pp. 276–291.