Exploiting Fine-Grain Parallelism in Transactional Database Systems

by

Cong Yan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author ..... Signature redacted .....
Department of Electrical Engineering and Computer Science
June 30, 2015

Certified by... Signature redacted .....
Daniel Sanchez
Assistant Professor
Thesis Supervisor

Accepted by ..... Signature redacted .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Exploiting Fine-Grain Parallelism in Transactional Database Systems

by

Cong Yan

Submitted to the Department of Electrical Engineering and Computer Science
on June 30, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Current database engines designed for conventional multicore systems exploit a fraction of the parallelism available in transactional workloads. Specifically, database engines only exploit *inter-transaction* parallelism: they use speculation to concurrently execute multiple, potentially-conflicting database transactions while maintaining atomicity and isolation. However, they do not exploit *intra-transaction* parallelism: each transaction is executed sequentially on a single thread. While fine-grain intra-transaction parallelism is often abundant, it is too costly to exploit in conventional multicores. Software would need to implement fine-grain speculative execution and scheduling, introducing prohibitive overheads that would negate the benefits of additional intra-transaction parallelism.

In this thesis, we leverage novel hardware support to design and implement a database engine that effectively exploits both inter- and intra-transaction parallelism. Specifically, we use *Swarm*, a new parallel architecture that exploits fine-grained and ordered parallelism. *Swarm* executes tasks speculatively and out of order, but commits them in order. Integrated hardware task queueing and speculation mechanisms allow *Swarm* to speculate thousands of tasks ahead of the earliest active task and reduce task management overheads. We modify Silo, a state-of-the-art in-memory database engine, to leverage *Swarm*'s features. The resulting database engine, which we call *SwarmDB*, has several key benefits over Silo: it eliminates software concurrency control, reducing overheads; it efficiently executes tasks within a database transaction in parallel; it reduces conflicts; and it reduces the amount of work that needs to be discarded and re-executed on each conflict.

We evaluate *SwarmDB* on simulated *Swarm* systems of up to 64 cores. At 64 cores, *SwarmDB* outperforms Silo by 6.7× on TPC-C and 6.9× on TPC-E, and achieves near-linear scalability.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor

Acknowledgments

First of all, I would like to thank Professor Daniel Sanchez for offering an opportunity to do research in his group. I am grateful for his patience all the time, and for always guiding me down the right path. He introduced me to many interesting topics in computer systems and architecture, letting me know that there is a big world there that I can learn and contribute to. His broad knowledge in computer architecture and rigorous manner in conducting experiments affected and inspired me a lot through this work.

I would also like to thank my collaborators, Mark Jeffrey and Suvinay Subramanian, with whom the working time has always been pleasant. I benefited a lot from discussions with them. Mark has helped me learn all the tools needed for this project. His working style shows me what a great computer engineer is like.

Then, my friends, and my family, who give me comfort, support and love. When I was down, confused and being poorly prepared for the challenges here, they have always been with me.

Contents

List of Figures

Chapter 1

Introduction

Multicore architectures are now pervasive, and increasing the number of cores has become the primary way to improve system performance. However, thread-level parallelism is often scarce [9,19], and many applications fail to make use of the growing number of cores. Thus, it is crucial to explore new architectural mechanisms to efficiently exploit as many types of parallelism as possible.

In this thesis, we focus on parallel transactional databases. Current database engines exploit thread-level parallelism by running multiple database transactions in parallel, using some form of speculative execution (concurrency control) to preserve transactional semantics. However, each transaction is executed sequentially on a single thread. In other words, these databases exploit coarse-grain inter-transaction parallelism but do not exploit fine-grain, intra-transaction parallelism. While fine-grain intra-transaction parallelism is often abundant, it is too costly to exploit in conventional multicores. Software would need to implement fine-grain speculative execution and scheduling, introducing prohibitive overheads that would negate the benefits of additional parallelism [17,18].

Targeting at this kind of fine-grained, speculative parallelism, our research group is designing a novel parallel architecture, called *Swarm*, with the following key features [21]:

1. *Task-based execution:* The architecture is optimized to execute short, atomic, timestamped tasks, as small as a few tens of instructions each. Hardware task queues implement speculative task creation and dispatch, drastically reducing task management overheads.
2. *Out-of-order execution with a large speculative window:* Tasks execute speculatively and out of order, but commit in order. To uncover enough parallelism, *Swarm* can run a task even if its parent task has yet to commit. *Swarm* can practically support thousands of outstanding speculative tasks, providing a large

task window from which to mine parallelism.

3. *Data-centric execution:* Data are divided among tiles, and tasks are sent to other tiles to run close to their data whenever possible, improving locality and reducing conflicts.

We present *SwarmDB*, a database system built atop of *Swarm* that exploits fine-grain parallelism with low overheads. In *SwarmDB*, each database transaction is decomposed into many small ordered tasks to exploit intra-transaction parallelism. Typically, each task reads or writes a single database tuple. Tasks from different transactions use disjoint timestamp ranges to preserve atomicity. This exposes significant fine-grain parallelism within and across transactions.

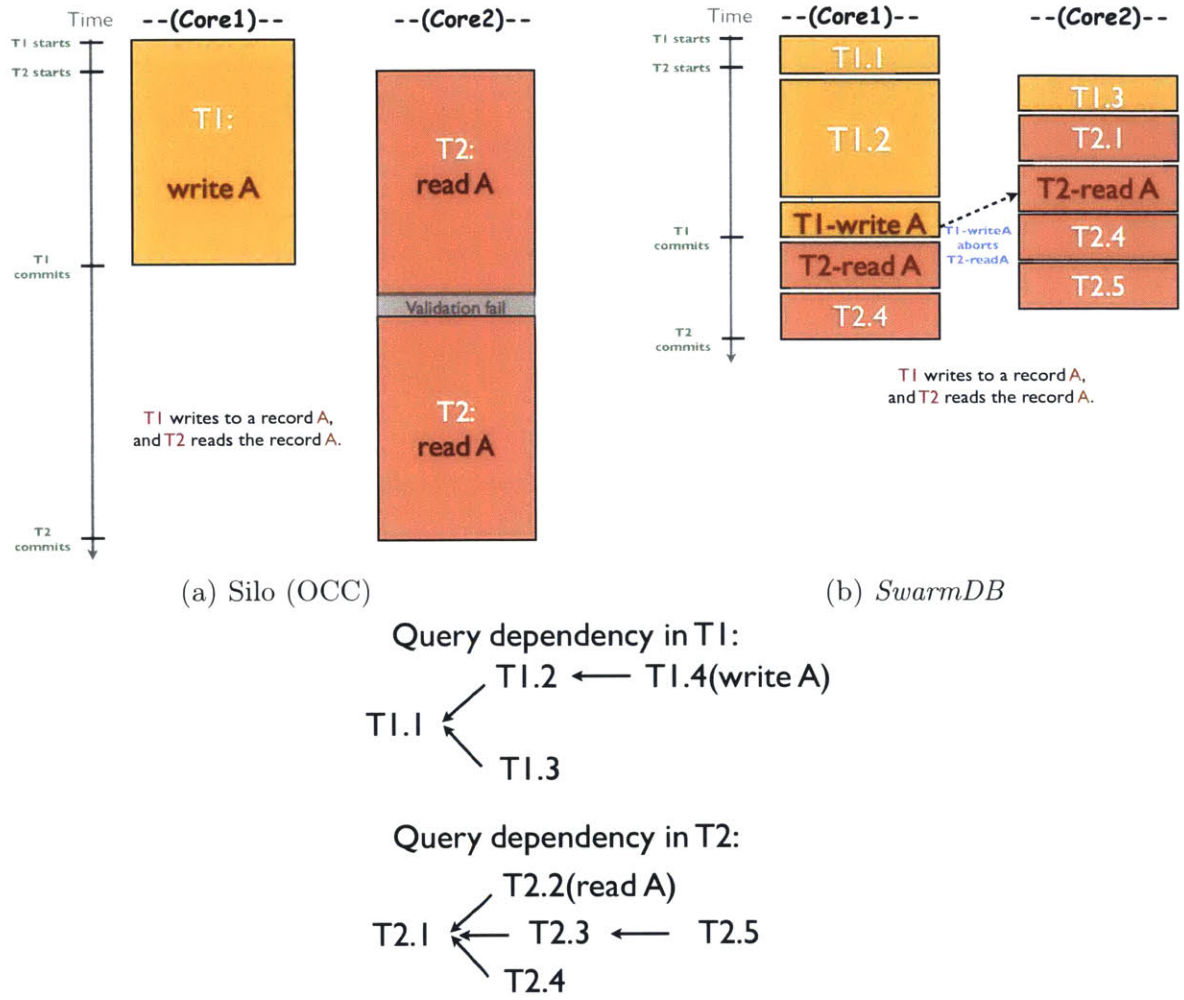
SwarmDB confers four key benefits over conventional on-line transaction processing (OLTP) database engines:

1. *Lower conflict detection and transactional execution overheads:.* Most concurrency control schemes are software-based, and incur significant overheads. Harizopoulos et al. [16] show that locking and latching in database systems contribute 30% of total transaction execution time. *SwarmDB* delegates concurrency control to hardware, drastically reducing these overheads, which improves performance even when parallelism is plentiful.
2. *Faster transaction execution:* Tasks within a single transaction can execute in parallel on multiple cores, rather than sequentially, improving transaction latency.
3. *Reduced conflicts and increased parallelism:.* Because each transactions uses several cores, there are fewer concurrent transactions than traditional in databases. When transactions are likely to conflict, fewer overlapped transactions reduces conflicts and aborted work.
4. *Reduced amount of wasted work per conflict:* In conventional database engines, conflicts result in aborts of the full database transaction. In contrast, on a conflict, *Swarm* selectively aborts only the mispeculated task and its dependents, so independent work within a transaction is not aborted.

Figure 1-1 shows an example execution schedule of *SwarmDB*, showcasing these benefits.

We evaluate *SwarmDB*, against Silo, a high-throughput in-memory database system, on three popular OLTP workloads: Bidding, TPC-C, and TPC-E. These workloads show different characteristics, and showcase the different advantages of *SwarmDB*. Bidding consists of short transactions without infrequent data conflicts, so benefits come from lower overheads. TPC-C has larger transactions with significant intra-transaction parallelism, and it inter-transaction parallelism depends on the data

size (number of warehouses). TPC-E includes very long transactions, and is more challenging to scale. At 64 cores, *SwarmDB* achieves 38×–58× self-relative speedups on these benchmarks, and outperforms Silo by 5% on Bidding, 6.7× on TPC-C with 4 warehouses, and 6.9× on TPC-E.



(a) Silo (OCC)

(b) *SwarmDB*

(c) Data dependency in T1 and T2

Figure 1-1: Example comparing Silo (timestamp-based optimistic concurrency control) and *SwarmDB* on a workload with two conflicting transactions (T1 and T2) on a two-core system. (a) In Silo, a read-write conflict causes T1 to abort and reexecute, incurring serialization and wasted work; (b) in contrast, *SwarmDB* executes both T1 and T2 across both cores, exploiting intra-transaction parallelism, and reduces the amount of wasted work on conflicts (only T2.2 is aborted and reexecuted); (c) *SwarmDB* decomposion of T1 and T2 into four and five dependent, ordered tasks.

Chapter 2

Background

2.1 Hardware Support for Fine-Grain and Speculative Parallelism

Prior work has investigated two main approaches to exploit speculative parallelism: thread-level speculation (TLS) and hardware transactional memory (HTM).

TLS: TLS schemes ship tasks from function calls or loop iterations to different cores, run them speculatively, and commit them in program order. Although TLS schemes support ordered speculative execution, two key problems prevent them from exploiting ordered irregular parallelism:

1. False data dependences limit parallelism: To run under TLS, ordered algorithms must be expressed as sequential programs, but their sequential implementations have limited parallelism.

   The root problem is that loops and method calls, the control-flow constructs supported by TLS schemes, are insufficient to express the order constraints among these tasks. By contrast, *Swarm* implements a more general execution model with timestamp-ordered tasks to avoid software queues, and implements hardware priority queues integrated with speculation mechanisms, avoiding spurious aborts due to queue-related references.
2. Scalability bottlenecks: Although prior TLS schemes have developed scalable versioning and conflict detection schemes, two challenges limit their performance with large speculation windows and small tasks:
   (a) Forwarding vs selective aborts: Most TLS schemes find it is desirable to forward data written by an earlier, still speculative task to later reader tasks. This prevents later tasks from reading stale data, reducing mispeculations on

tight data dependences. However, it creates complex chains of dependences among speculative tasks. Thus, upon detecting mispeculation, most TLS schemes abort the task that caused the violation *and all later speculative tasks* [11, 14, 32, 35, 36].

We find that forwarding speculative data is crucial for *Swarm*. However, while aborting all later tasks is reasonable with small speculative windows (2–16 tasks are typical in prior work), *Swarm* has a 1024-task window, and unselective aborts are impractical. To address this, we contribute a novel conflict detection scheme based on eager version management that allows both forwarding speculative data and selective aborts of dependent tasks.

(b) Commit serialization: Prior TLS schemes enforce in-order commits by passing a token among ready-to-commit tasks [14, 32, 35, 36]. Each task can only commit when it has the token, and passes the token to its immediate successor when it finishes committing.

This approach cannot scale to the commit throughput that *Swarm* needs. For example, with 100-cycle tasks, a 64-core system should commit 0.64 tasks/cycle on average. Even if commits were instantaneous, the latency incurred by passing the token makes this throughput unachievable.

HTM: HTM schemes use speculation to guarantee that certain code regions in multithreaded programs, called transactions, execute atomically and in isolation. However, it is difficult to use HTM to fully exploit the parallelism in the application, especially for long transactions which include large footprints of reads and writes. Prior work [7, 24, 26] has investigated several methods to support nested transactions in HTM. Nested transactions improve the performance of HTM by exploiting more parallelism. Closed-nested transactions extend isolation of an inner transaction until the top-level transaction commits. Implementations may flatten nested transactions into the top-level one, resulting in a complete abort on conflict, or allow partial abort of inner transactions [28]. Open-nested transactions allow a committing the inner transaction to immediately release isolation, which increases parallelism at the cost of both software and hardware complexity [28]. Supporting nested transactions requires changes on program interface since the programmer needs to specify the region of nested transaction. The performance gain depends on the way transactions are nested, which relies heavily on how the programmer writes the code. For long and complicated transactions, this requires large effort.

Even though *Swarm* is designed to support ordered parallelism, it still has comparable performance to HTM in unordered algorithm semantics, and has simpler program

interface than HTM which supports nested transactions. Since *Swarm* commits tasks in the order specified, all tasks are fully isolated appearing to the programmer, and there is no need to specify different nesting levels, which makes programming much simpler comparing to HTM.

2.2 Database Systems on TLS and TM Hardware

Prior work has used hardware techniques to improve the performance of database workloads. Colohan et al. [5,6] apply TLS to to exploit intra-transaction parallelism in database workloads. Compared to this work, we instead expose many finer-grain tasks, avoid ordering sibling tasks that are known independent, and extract inter-transaction parallelism by speculating many tasks ahead (running tasks from several transactions concurrently if needed).

Beside TLS, HTM is also a choice for improving performance of database systems. Wang et al. [42] built a database system on Intel's restricted transactional memory (RTM), which outperforms Silo by 52%. HTM is efficient in handling conflict checks, which eliminates the large overhead caused by software latching or locking in traditional database systems. However, since HTM uses caches to track transaction's read/write set, the transaction is forced to abort when its data size exceeds the cache size. Although HTM reduces overheads, it does not reduce the abort rate of traditional concurrency control schemes and is subject to the same scalability bottlenecks.

2.3 State-of-the-art Software Concurrency Control Schemes

Prior work has proposed a wide variety of concurrency control schemes for database systems. Widely studied schemes include two-phase locking (2PL), optimistic concurrency control (OCC), and multi-version concurrency control (MVCC). However, these schemes either limit concurrency or add excessive overheads. They usually have a tradeoff between the software code complexity and parallelism they exploit. 2PL grabs a tuple lock upon first accessing of the tuple and releases the lock at commit, which is much longer than necessary. A transaction has to wait for another transaction to finish before actually starting if it conflicts at the first tuple in the transaction. OCC checks the read/write set intersection before commit. The locking range is short since it only needs to lock the write set, but it redoes the complete transaction on abort even if only one tuple in the transaction conflicts, which causes much wasted work.

MVCC doesn't acquire lock; but software-managed versioning has very large overhead, and the whole transaction gets aborted when timestamp sequence on one tuple is violated. In addition, all the above concurrency control schemes redo the complete transaction that is aborted, while only several queries of the transaction has data conflict.

None of the above schemes exploit intra-transaction parallelism. In distributed databases, independent queries within a transaction are executed in parallel. The data is distributed in different machines, so sending queries to where the data is located is an obvious choice. However, such parallelism is hardly explored in in-memory databases due to the large overhead of sending queries to different cores. The performance loss is significant especially for long transactions.

SwarmDB doesn't use software concurrency control, leveraging *Swarm* instead to ensure the consistency and atomicity of transactions. Each transaction has a timestamp range and each task within a transaction is assigned a unique timestamp. The in-order commit mechanism of *Swarm* ensures that all the transactions appear to be executed sequentially. *SwarmDB* naturally exploits intra-transaction parallelism.

Chapter 3

Swarm: An Architecture for Ordered Parallelism

This chapter introduces *Swarm*, the architecture on which *SwarmDB* is built. For more details, see [21].

Figure 3-1 shows *Swarm*'s high-level organization. *Swarm* is a tiled CMP. Each tile has a group of simple cores. Each core has small, private, write-through L1 caches. All cores in a tile share a per-tile L2 cache, and each tile has a slice of a shared NUCA L3 cache. Each tile features a *task unit* that queues, dispatches, and commits tasks. Tiles communicate through a mesh NoC.

Key features: *Swarm* is optimized to execute short tasks with programmer-specified order constraints. Programmers define the execution order by assigning *timestamps* to tasks. Tasks can create children tasks with equal or higher timestamp than their own. Tasks appear to execute in global timestamp order, but *Swarm* uses speculation to elide order constraints.

Swarm is coherently designed to support a large speculative task window efficiently. *Swarm* has no centralized structures: each tile's task unit queues runnable, non-speculative tasks, and maintains the speculative state of finished tasks that cannot yet commit. Task units only communicate when they send new tasks to each other to maintain load balance, and, infrequently, to determine which finished tasks can be committed.

Swarm speculates far ahead of the earliest active task, and runs tasks even if their parent is still speculative. Figure 3-2(a) shows this process: a task with timestamp 0 is still running, but tasks with higher timestamps and several speculative ancestors are running or have finished execution. For example, the task with timestamp 51, currently running, has three still-speculative ancestors, two of which have finished and

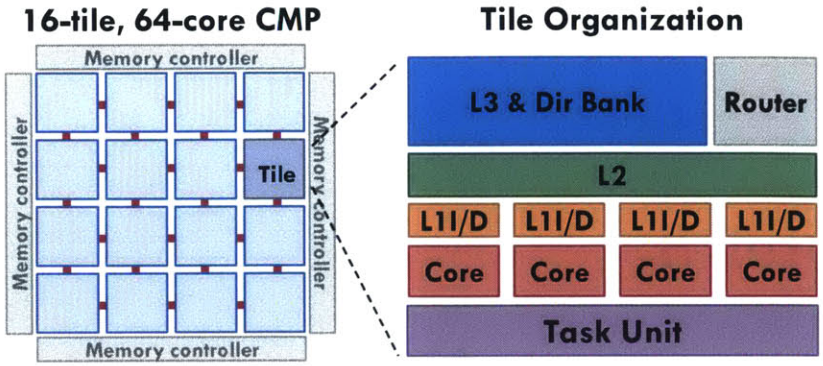
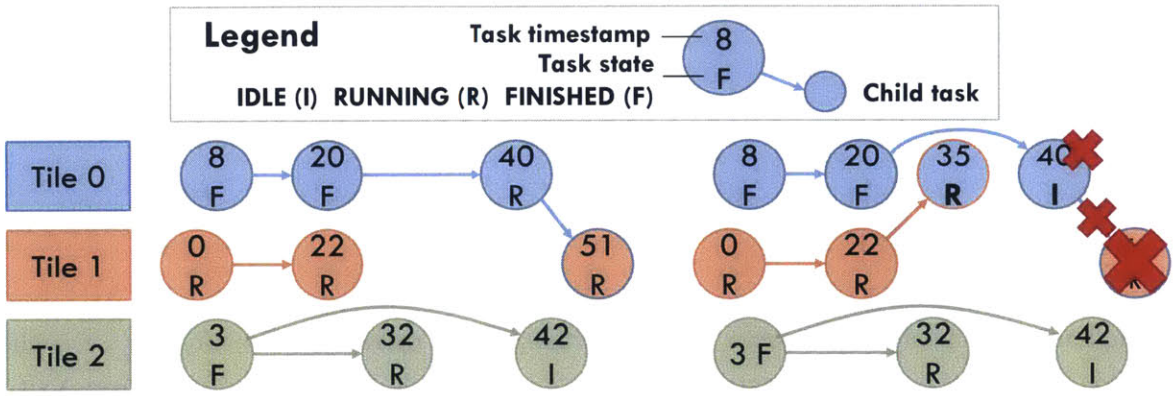


Figure 3-1: Target tiled CMP and tile configuration. Duplicated from [21].



(a) Tasks far ahead of the minimum timestamp (0) are run, even when parent is still speculative

(b) Selective aborts: Task 35 aborts 40 and child 51, but not 42

Figure 3-2: Example of deep speculation and data-centric execution. By executing tasks even if their parents are speculative, deep speculation uncovers ordered parallelism, but may cause cascading aborts.

are waiting to commit (8 and 20) and one which is still running (40).

Allowing tasks with speculative ancestors to execute uncovers significant parallelism, but may induce aborts that span multiple tasks. For example, in Figure 3-2(b) a new task with timestamp 35 conflicts with task 40, so 40 is aborted and child task 51 is both aborted and discarded. These aborts are *selective*, and only affect tasks whose speculative ancestors are aborted, or tasks that have read data written by an aborted task.

3.1 ISA Extensions and Programming Model

Swarm manages and dispatches tasks using hardware task queues. A task is represented by a descriptor with the following architectural state: the task's function pointer, a 64-bit timestamp, and the task's arguments.

Tasks appear to run in timestamp order. Tasks with the same timestamp may execute in any order, but run *atomically*—the system lazily selects an order for them.

A task can create one or more *children tasks* with equal or higher timestamp than its own timestamp. A child is ordered after its *parent*, but children with the same timestamp may execute in any order. Because hardware must track parent-child relations, tasks may create a limited number of children (8). Tasks that need more children enqueue a single task that creates them.

Swarm adds instructions to enqueue and dequeue tasks. The `enqueue_task` instruction accepts a task descriptor (held in registers) as its input and queues the task for execution. A thread uses the `dequeue_task` instruction to start executing some previously-enqueued task by making the timestamp and arguments available to the task (in registers), and initiating speculative execution at the task's function pointer. Task execution ends with a `finish_task` instruction.

`dequeue_task` stalls the core if an executable task is not immediately available, avoiding busy-waiting. When no tasks are left in any task unit and all threads are stalled on `dequeue_task`, the algorithm has terminated, and `dequeue_task` jumps to a configurable pointer to handle termination.

API: We design a low-level C++ API that uses these mechanisms. Tasks are simply functions with signature:

```
void taskFn(timestamp, args...)
```

Code can enqueue other tasks by calling:

```
enqueueTask(taskFn, timestamp, args...)
```

3.2 Task Queuing and Prioritization

The task unit has two main structures:

1. The *task queue* holds task descriptors (function pointer, timestamp, and arguments).
2. The *commit queue* holds the speculative state of tasks that have finished execution but cannot yet commit.

Figure 3-3 shows how these queues are used throughout the task's lifetime. Each new task allocates a task queue entry, and holds it until commit time. Each task allocates a commit queue entry when it finishes execution, and also deallocates it at commit time. For now, assume these queues always have free entries.

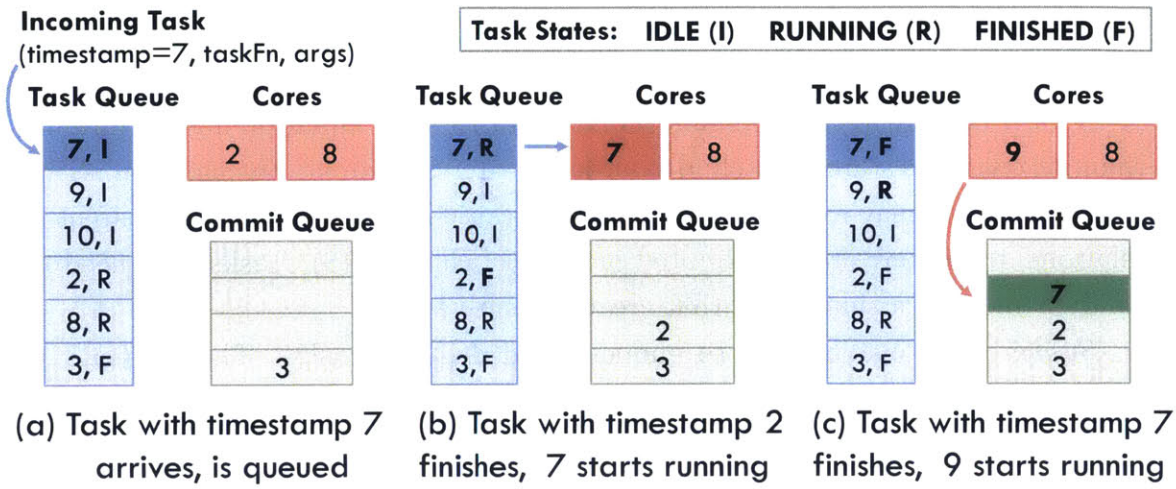


Figure 3-3: Task queue and commit queue utilization through a task's lifetime. Duplicated from [21].

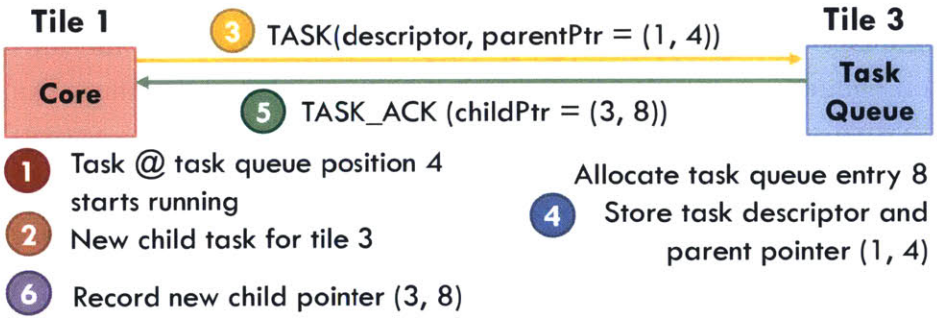


Figure 3-4: Task creation protocol. Cores send new tasks to other tiles for execution. To track parent-child relations, parent and child keep a pointer to each other.

Together, the task queue and commit queue are similar to a reorder buffer, but at task-level rather than instruction-level. They are separate structures because commit queue entries are larger than task queue entries, and typically fewer tasks wait to commit than to execute. However, unlike in a reorder buffer, tasks do not arrive in priority order. Both structures manage their free space with a freelist and allocate entries independently of task priority order, as shown in Figure 3-3.

Task enqueues: When a core creates a new task (through `enqueue_task`), it sends the task to a randomly-chosen target tile following the protocol in Figure 3-4. Parent and child track each other using *task pointers*. A task pointer is simply the tuple $(tile, task\ queue\ position)$. This tuple uniquely identifies a task because it stays in the same task queue position throughout its lifetime.

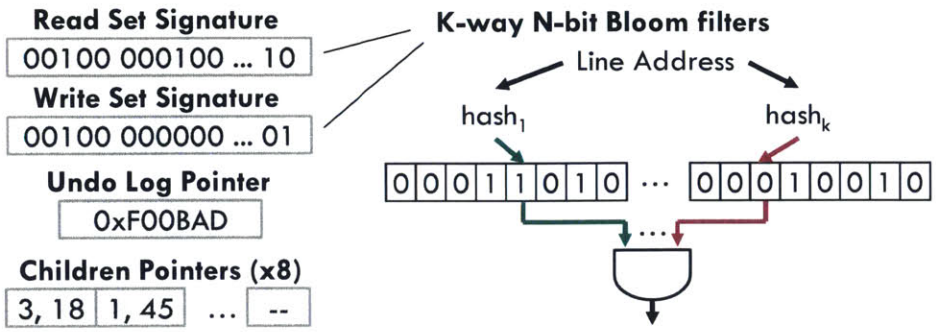


Figure 3-5: Speculative state for each task. Each core and commit queue entry maintains this state. Read and write sets are implemented with space-efficient Bloom filters. Duplicated from [21].

Task prioritization: Tasks are prioritized for execution in timestamp order. When a core calls `dequeue_task`, the highest priority idle task in the task queue is selected for execution. Since task queues do not hold tasks in priority order, an auxiliary *order queue* is used to find this task.

The order queue can be cheaply implemented with two small ternary content-addressable memories (TCAMs) with as many entries as the task queue (e.g., 256), each of which stores a 64-bit timestamp. With Panigrahy and Sharma's PIDR_OPT method [29], finding the next task to dispatch requires a single lookup in both TCAMs, and each insertion (task creation) and deletion (task commit or squash) requires two lookups in both TCAMs. SRAM-based implementations are also possible, but we find the small TCAMs to have a moderate cost.

3.3 Speculative Execution and Versioning

The key requirements for speculative execution in *Swarm* are allowing fast commits and a large speculative window. To this end, we adopt *eager versioning*, storing speculative data in place and logging old values. Eager versioning makes commits fast, but aborts are slow. However, *Swarm*'s execution model makes conflicts rare, so eager versioning is the right tradeoff.

Eager versioning is common in hardware transactional memories [27], which do not perform ordered execution or speculative data forwarding. By contrast, most TLS systems use lazy versioning (buffering speculative data in caches) or more expensive multiversioning [3, 14, 15, 30, 31, 32, 35, 36, 37] to limit the cost of aborts. Few, early TLS schemes are eager [1, 11], and they still suffer from the limitations.

Swarm's speculative execution borrows from LogTM and LogTM-SE [27, 43]. Our key contributions over these and other speculation schemes are (i) conflict-

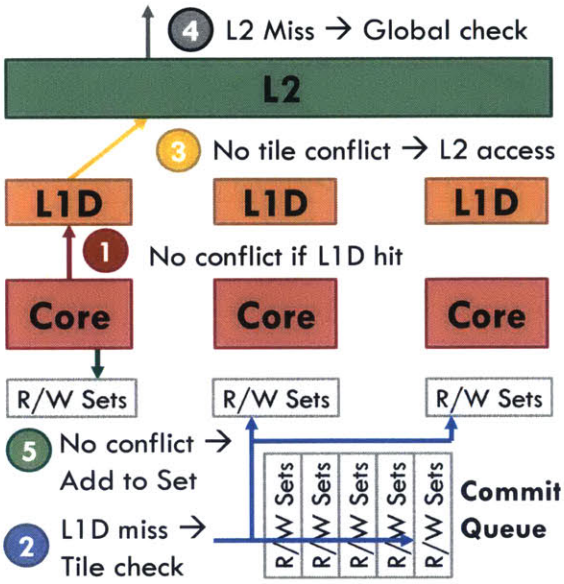


Figure 3-6: Local, tile, and global conflict detection for an access that misses in the L1 and L2.

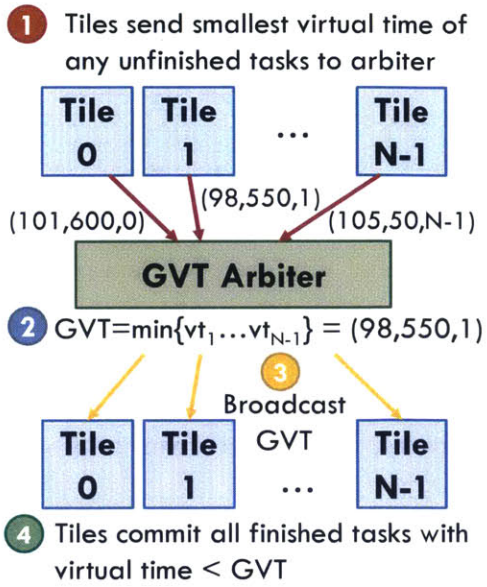


Figure 3-7: Global virtual time (GVT) update protocol, modified from [21].

detection (section 3.4) and selective abort techniques (section 3.5) that leverage *Swarm*'s hierarchical memory system and Bloom filter signatures to scale to large speculative windows, and (ii) a technique that exploits *Swarm*'s large commit queues to achieve high-throughput commits (section 3.6).

Figure 3-5 shows the per-task state needed to support speculation: read- and write-set signatures, an undo log pointer, and child pointers. Each core and commit queue entry hold this state.

A successful `dequeue_task` instruction jumps to the task's code pointer and initiates speculation. Since speculation happens at the task level, there are no register checkpoints, unlike in HTM and TLS. Like in LogTM-SE, as the task executes, hardware automatically performs conflict detection on every read and write (section 3.4). Then, it inserts the read and written addresses into the Bloom filters, and, for every write, it saves the old memory value in a memory-resident undo log. Stack addresses are not conflict-checked or logged.

When a task finishes execution, it allocates a commit queue entry; stores the read and write set signatures, undo log pointer, and children pointers there; and frees the core for another task.

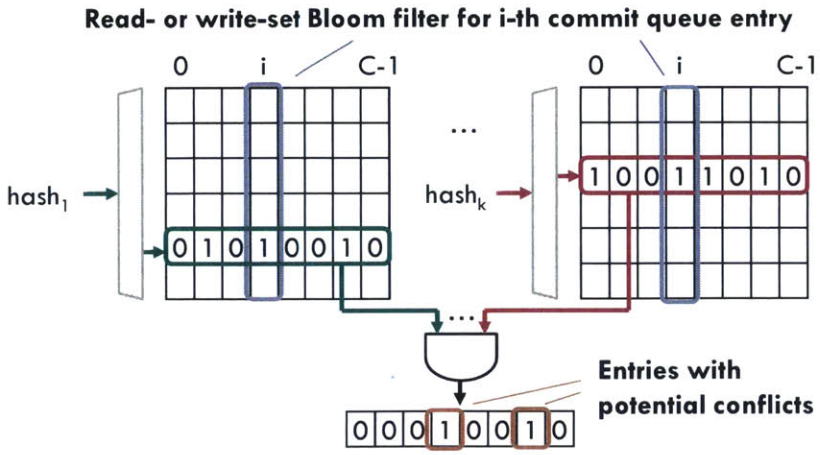


Figure 3-8: Commit queues store read- and write-set Bloom filters by columns, so a single word read retrieves one bit from all entries. All entries are then checked in parallel.

3.4 Virtual Time-Based Conflict Detection

Conflict detection is based on a priority order that respects both programmer-assigned timestamps and parent-child relationships. Conflicts are detected at cache line granularity.

Unique virtual time: Tasks may have the same programmer-assigned timestamp. However, conflict detection has much simpler rules if tasks have a total order. Therefore, tasks are given a *unique virtual time* when they are dequeued for execution. Unique virtual time is the 128-bit tuple (programmer timestamp, dequeue cycle, tile id). Conflicts are resolved using this unique virtual time, which tasks preserve until they commit.

Unique virtual times incorporate the ordering needs of programmer-assigned timestamps and parent-child relations: children always start execution after their parents, so a parent always has a smaller dequeue cycle than its child, and thus a smaller unique virtual time, even when parent and child have the same timestamp.

Conflicts and forwarding: Conflicts arise when a task accesses a line that was previously accessed by a higher-virtual time task. Suppose two tasks, t_1 and t_2, are running or finished, and t_2 has a higher virtual time. A read of t_1 to a line written by t_2 or a write to a line read or written by t_2 causes t_2 to abort. However, t_2 can access data written by t_1 even if t_1 is still speculative. Thanks to eager versioning, t_2 automatically uses the latest copy of the data—there is no need for speculative data forwarding logic [11].

Hierarchical conflict detection: *Swarm* exploits the cache hierarchy to reduce conflict checks. Figure 3-6 shows the different types of checks performed in an access:

1. The L1 is managed to ensure L1 hits are conflict-free.
2. L1 misses are checked against other tasks in the tile (both in other cores and in the commit queue).
3. L2 misses, or L2 hits where a virtual time check (described below) fails, are checked against tasks in other tiles. As in LogTM [27], the L3 directory uses memory-backed *sticky bits* to only check tiles whose tasks may have accessed the line. Sticky bits are managed exactly as in LogTM.

Any of these conflicts trigger task aborts.

Using caches to filter checks: The key invariant that allows caches to filter checks is that, when a task with virtual time T installs a line in the (L1 or L2) cache, that line has no conflicts with tasks of virtual time $> T$. As long as the line stays cached with the right coherence permissions, it stays conflict-free. Because conflicts happen when tasks access lines out of virtual time order, if another task with virtual time $U > T$ accesses the line, it is also guaranteed to have no conflicts.

However, accesses from a task with virtual time $U < T$ must trigger conflict checks, as another task with intermediate virtual time X, $U < X < T$, may have accessed the line. U's access does not conflict with T's, but may conflict with X's. For example, suppose task with virtual time $X = 2$ writes line A. Then, task $T = 3$ in another core reads A. This is not a conflict with X's write, so A is installed in T's L1. The core then finishes T and dequeues a task $U = 1$ that reads A. Although A is in the L1, U has a conflict with X's write.

We handle this issue with two changes. First, when a core dequeues a task with a smaller virtual time than the one it just finished, it flushes the L1. Because L1s are small and write-through, this is fast, simply requiring to flash-clear the valid bits. Second, each L2 line has an associated *canary virtual time*, which stores the lowest task virtual time that need not perform a global check. For efficiency, lines in the same L2 set share the same canary virtual time. For simplicity, this is the maximum virtual time of the tasks that installed each of the lines in the set, and is updated every time a line is installed.

Efficient commit queue checks: To allow large commit queues (e.g., 64 tasks/queue), conflict checks must be efficient. To this end, we leverage that checking a K-way Bloom filter only requires reading one bit from each way. As shown in Figure 3-8, Bloom filter ways are stored in columns, so a single 64-bit access per way

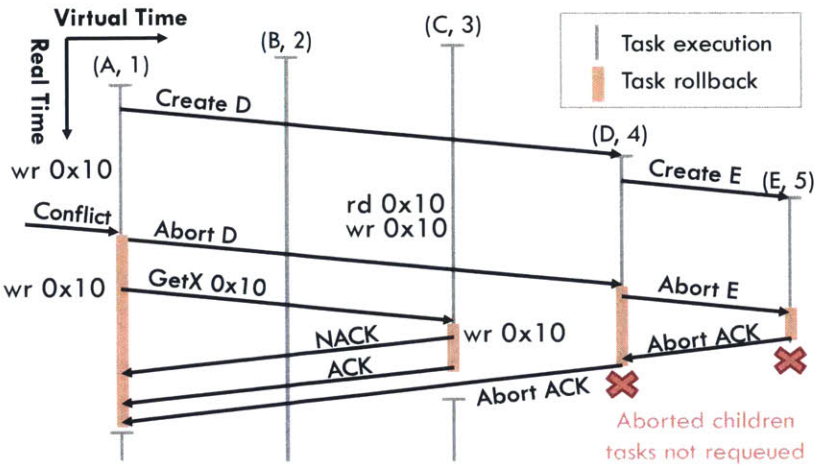


Figure 3-9: Selective abort protocol. Suppose $(A, 1)$ must abort after it writes 0x10. $(A, 1)$'s abort squashes child $(D, 4)$ and grandchild $(E, 5)$. During rollback, A also aborts $(C, 3)$, which read A's speculative write to 0x10. $(B, 2)$ is independent and thus not aborted.

reads all the necessary bits. Reading and ANDing all ways yields a word that indicates potential conflicts. The virtual times of entries whose position in this word is 1 are checked.

3.5 Selective Aborts

Upon a conflict, *Swarm* aborts the later-timestamp task and all its dependents: its children and other tasks that have accessed data written by the aborting task. Hardware aborts each task t in two steps:

1. Notify t's children to abort and be removed from their task queues.
2. Walk t's undo log in LIFO order, writing old values. If one of these writes conflicts with a higher-virtual time task, wait for it to abort and continue t's rollback.
3. Clear t's signatures and free its commit queue entry.

Applied recursively, this procedure selectively aborts all dependent tasks, as shown in Figure 3-9. This scheme has two key benefits. First, it reuses the conflict-detection logic used in normal operation. Undo-log writes (e.g., A's second `wr 0x10` in Figure 3-9) are normal conflict-checked writes, issued with the tasks's timestamp to detect all later readers and writers. Second, this scheme does not explicitly track data dependences among tasks. Instead, uses the conflict-detection protocol to recover them as needed. This is important, because any task may have served speculative data to many other tasks, with would make tracking state expensive. For example,

tracking all possible dependences on a 1024-task window using bit-vectors would require $1024 \times 1023 \simeq 1$ Mbit of state.

3.6 Scalable Ordered Commits

To achieve high-throughput commits, *Swarm* adapts the virtual time algorithm [20], common in parallel discrete event simulation [10]. Figure 3-7 shows this protocol. Tiles periodically send the smallest unique virtual time of any unfinished (running or idle) task to an arbiter. Idle tasks do not yet have a unique virtual time and use (timestamp, current cycle, tile id) for the purposes of this algorithm. The arbiter computes the minimum virtual time of all unfinished tasks, called the *global virtual time* (GVT), and broadcasts it to tiles. To preserve ordering, only tasks with virtual time $<$ GVT can commit.

The key insight is that, by combining the virtual time algorithm with *Swarm*'s large commit queues, *commit costs are amortized over many tasks.* A single GVT update often causes many finished tasks to commit. For example, if in Figure 3-7 the GVT jumps from (80,100,2) to (98,550,1), all tasks with virtual time (80,100,2)$<$ t $<$(98,550,1) can commit. GVT updates happen sparingly (e.g., every 200 cycles) to limit bandwidth. Less frequent updates reduce bandwidth but increase commit queue occupancy.

In addition, eager versioning makes commits fast: a task commits by freeing its task and commit queue entries, a single-cycle operation. Thus, if a long-running task holds the GVT for some time, once it finishes, commit queues quickly drain and catch up to execution.

Compared with prior TLS schemes that use successor lists and token passing to reconcile order, this scheme does not even require finding the successor and predecessor of each task, and does not serialize commits.

For the system sizes we evaluate, a single GVT arbiter suffices. Larger systems may need a hierarchy of arbiters.

3.7 Handling Limited Queue Sizes

The per-tile task and commit queues may fill up, requiring a few simple actions to ensure correct operation.

Task queue virtualization: Applications may create an unbounded number of tasks and schedule them for a future time. *Swarm* uses an overflow/underflow

mechanism to give the illusion of unbounded hardware task queues [13, 23, 33]. When the per-tile task queue is nearly full, the task unit dispatches a special, non-speculative *coalescer* task to one of the cores. This coalescer task removes several *non-speculative*, idle task descriptors with high programmer-assigned timestamps from the task queue, stores them in memory, and enqueues a *splitter* task that will re-enqueue the overflowed tasks.

Virtual time-based allocation: The task and commit queues may also fill up with speculative tasks. The general rule to avoid deadlock due to resource exhaustion is to always prioritize lower-virtual time tasks, aborting other tasks with higher virtual times if needed. For example, if a tile speculates far ahead, fills up its commit queue, and then receives a task that precedes all other speculative tasks, the tile must let the preceding task execute to avoid deadlock. This results in three specific policies for the commit queue, cores, and task queue.

Commit queue: If task t finishes execution, the commit queue is full, and t precedes any of the tasks in the commit queue, it aborts the highest-virtual time finished task and takes its commit queue entry. Otherwise, t stalls its core, waiting for an entry.

Cores: If task t arrives at the task queue, the commit queue is full, and t precedes all tasks in cores, t aborts the highest-virtual time task and takes its core.

Task queue: Suppose task t arrives at a task unit but the task queue is full. If some tasks are non-speculative, a coalescer is running, and the task waits for a free entry. If all tasks in the task queue are speculative, the enqueued request is NACK'd and the *parent task* stalls, and retries the enqueue using linear backoff. To avoid deadlock, we leverage that when a task's unique virtual time matches the GVT, it is the smallest-virtual time task in the system, and cannot be aborted. This task need not keep track of its children (no child pointers), and when those children are sent to another tile, they can be overflowed to memory if the task queue is full. This ensures that the GVT task makes progress, avoiding deadlock.

Chapter 4

Building an In-Memory Database System on *Swarm*

4.1 Mapping Database Transactions to *Swarm* Tasks

We will design and implement an in-memory OLTP database that exploits the intra-transaction parallelism available with a decomposition of partially-ordered tasks. In this section, we present the intuition behind porting a multi-table transaction of single-table queries into several partially-ordered tasks.

We use the TPC-C payment transaction as an example. All operations in the transaction are written sequentially, and apart from transaction-level parallelism, these operations would be executed sequentially. However in this example, operations on distinct tables can be executed in parallel if they lack a data dependency. We naturally group operations on a single table into a task. To ensure that the queries/updates in a transaction looks like being executed sequentially, each task is assigned a unique timestamp according to their execution order in the transaction. Code below shows the code for payment transaction using *Swarm* API:

```
void modify_warehouse(args...){
  execute_query("SELECT...FROM warehouse WHERE ...");
  execute_query("UPDATE warehouse SET...");
}
void modify_district(args...){
  execute_query("SELECT...FROM district WHERE âĂę");
  execute_query("UPDATE district SET...");
}
void modify_customer(args...){
  execute_query("UPDATE customer SET...");
}
void insert_history(args...){
  execute_query("INSERT history ...");
}
```

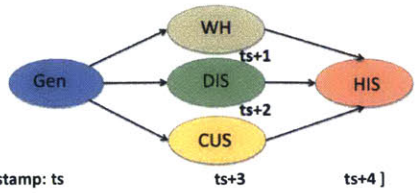


Figure 4-1: Data dependency among tasks in payment txn.

Figure 4-2: Execution sequence of 2 payment txns in a 2-core system.

```
void transaction(uint64_t ts){
  enqueueTask(modify_warehouse, ts, args...);
  enqueueTask(modify_district, ts+1, args...);
  enqueueTask(modify_customer, ts+2, args...);
  enqueueTask(insert_history, ts+3, args...);
}
```

Atomicity and consistency are guaranteed by assigning each database transaction a non-overlapping timestamp range. By doing that, transactions is forced upon a preliminary execution order. Queries within a transaction each claim a distinct timestamp, which reflects the order of queries in the transaction. Since tasks in *Swarm* is committed in order, the atomicity and consistency property of transaction will not be violated.

Figure 4-1 shows the data dependency among these tasks. If a task that depends on previous tasks and is executed earlier, it will be aborted. If a task has data conflict with another task, the one with earlier timestamp wins and the the other aborted. One possible execution sequence in 2-core environment under *Swarm* is shown in Figure 4-2.

Database on *Swarm* have further advantage comparing to database using conventional concurrency control when the number of available cores exceeds the number of available transactions. Some cores would be idle with conventional concurrency control, while all threads will be busy in *Swarm*. One possible execution sequence in a 4-core environment under *Swarm* is shown in Figure 4-3.

Since we are naturally mapping queries to tasks, the translation from traditional database system to *SwarmDB* is easy. A query parser/optimizer can handle the work of analyzing the data dependency among queries and determines the arguments for the tasks. Wrapping the query execution with our task API is straightforward.

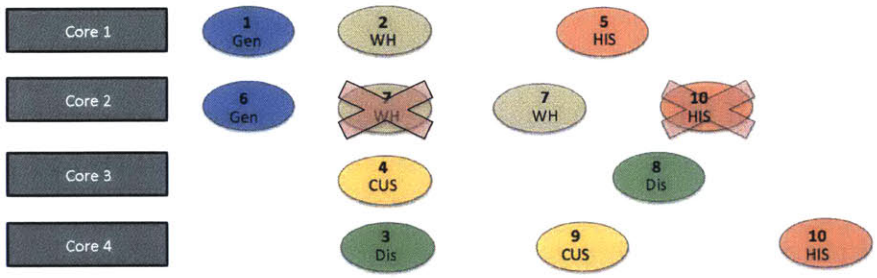


Figure 4-3: Execution sequence of 2 payment txns in a 4-core system.

Besides, by analyzing the longest possible query in an application, we can determine the timestamp range for each transaction in that application.

4.2 Application-specific Optimizations

4.2.1 Handling known data dependencies

Some applications has transactions in which queries show explicit data dependencies. For example, in the payment transaction mentioned above (Figure 4-1), the "INSERT history" query have data dependency on "SELECT warehouse", "SELECT customer" queries since the payment history item to be inserted contains the information of warehouse the payment is paid to, as well as the customer information who is doing the payment. If we issue the task executing "INSERT history" and task executing "SELECT warehouse" at the same time, the "INSERT history" task is very likely to get aborted, thus making the system doing useless work. However, we can avoid this by making the task of "SELECT warehouse" issuing the task "INSERT history" when it finishes the query, and pass the warehouse information as parameters to the task "INSERT history".

However, the "INSERT history" is still very likely to abort since it is hard to say whether the task "SELECT customer" will be executed earlier than "INSERT history". But we can set an indicator in the task of "INSERT history" to indicate how many of its parents finished execution, and it only starts when all of its parent tasks are done. The analysis of the explicit parent-child relationship can be easily done by query parser/query optimizer.

4.2.2 Data partition

Some application has long scan operations, like TPCE. Scans on large tables often take long time, and cause large amount of useless work if a single record is modified

by another concurrent transaction, causing the scan to abort.

To make the long scans execute more efficiently, we partition the table to allow scanning on partial tables go on in parallel and then merge the results. There is a tradeoff here: if we partition the table into smaller parts, the scan operations has more parallelism; however the task creation and merging overhead increases. Adjusting the number of partitions according to the cores available is a possible solution. We leave choosing an optimized partition algorithm to our future work.

4.3 Durability

A transaction is durable if all its modifications are recorded on durable storage and all transactions serialized before it are durable. Thus, durability requires the system to recover the effects of a set of transactions that form a prefix of the serial order [41]. In *SwarmDB*, we implemented value logging [44]. Logging is done by separate tasks. Although the execution of transaction is out-of-order, the logging needs to be ordered for the purpose of recovery after a crash.

To solve this problem, we created a space to store the log, and each query log hashes into a slot in this space using task timestamp as hash key. Logs are copied from the temporary memory to disk periodically. Timestamp for each query is attached to the log; when the logs are copied to disk, they're sorted according to their timestamp, so that logs appear to be sequential on disk. Since we also log transaction commit along with the timestamp, and all the transactions have distinct timestamp range, we can easily differentiate on-going transactions and finished transactions by reading the log, and only recover values written by finished transactions upon recovery.

By applying a hashing as 2-level universal hashing, only $\Omega(n)$ hashing space is needed, while n is the max number of query tasks on the fly. However currently we assume logging is cheap in space, so we give a sufficient number of slots for logging. Since the simulator for *Swarm* doesn't simulate disk latency, we don't plan to implement copying logging from memory to durable storage. However, a background process can handle the work of moving the log from the hash space to the disk, without affecting the performance of task threads.

Chapter 5

Evaluation

We first report *SwarmDB*'s speedups and performance against Silo. We then analyze *SwarmDB*'s behavior in depth and study its sensitivity to various architectural parameters.

5.1 *SwarmDB* Benchmarks

We use Silo as our baseline, and evaluate *SwarmDB* on a variety of commonly used OLTP benchmarks.

Bidding benchmark: This is a small benchmark simulating the situation where many bidders participating in a bidding of many items. It consists of three tables and one type of transaction. Each transaction includes at most five queries, so transactions are short. If there're many users placing bids on many different items, there're barely conflicts among concurrent transactions. Since *SwarmDB* is gaining benefit from exploiting intra-transaction parallelism and doing clever partial abort, the bidding benchmark with small transaction and light conflict presents the worst case for *SwarmDB*.

TPC-C benchmark: The TPC-C benchmark is the current industry standard for evaluating the performance of OLTP systems [38]. It consists of nine tables and five types of transactions that simulate a warehouse-centric order processing application. All of the transactions in TPC-C provide a warehouse id as an input parameter that is the ancestral foreign key for all but one of TPC-C's tables [8]. This means that the data can be perfectly partitioned by warehouses. Based on this characteristics, we evaluate *SwarmDB* against Silo on two sets of experiments: strong scalability test and weak scalability test (explained in section 5.2).

TPC-C represents OLTP benchmark with long transaction and complicated data dependencies within transaction, as well as adjustable data conflict rate (which is

different in two experiment sets). Improving TPC-C performance under high data contention usually requires dynamically tracking data dependencies and breaking transactions, which is very challenging in traditional database systems.

TPC-E benchmark: The TPC-E benchmark models a financial brokerage house. There are three components: customers, brokerage house, and stock exchange. TPC-E's focus is the database system supporting the brokerage house, while customers and stock exchange are simulated to drive transactions at the brokerage house [4]. Comparing to TPC-C, the TPC-E benchmarks shows poor spatial and temporal locality. In our experiments, we only use a combination of five most frequent transactions in TPC-E (section 5.2).

The TPC-E benchmark is consist of long transactions containing long queries. Scans and joins on large tables are very frequent in this benchmark, which makes the overhead of concurrency control excessive [39]: for locking based algorithm, scans and joins means a large number of locks to be hold by one transaction, which complicates the locking management; for read-write set checking algorithm (like OCC), the cross checking of big read/write set slows down the transaction; for timestamp based algorithm, long transaction means the large span over timestamp, making it likely to have many aborts due to timestamp violation.

5.2 Methodology

5.2.1 System configuration

We use an in-house microarchitectural, event-driven, sequential simulator based on Pin [25] to model a 64-core CMP with a 3-level cache hierarchy. We use IPC-1 cores with detailed timing models for caches, on-chip network, main memory, and *Swarm* features (e.g., conflict checks, aborts, etc.). 5.1 details the modeled configuration.

Idealized memory allocation: Dynamic memory allocation is not simulated in detail, and a scalable solution is left to future work. In principle, we could build a task-aware allocator with per-core memory pools to avoid serialization. However, building high performance allocators is complex [12, 34]. Instead, the simulator allocates and frees memory in a task-aware way. Freed memory is not reused until the freeing task commits to avoid spurious dependences. Each allocator operation incurs a 30-cycle cost. For fairness, serial and software-parallel implementations also use this allocator.

Cores	64 cores in 16 tiles (4 cores/tile), 2 GHz, x86-64 ISA, IPC-1 except misses and *Swarm* instructions
L1 caches	16 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	256 KB, per-tile, 8-way, inclusive, 7-cycle latency
L3 cache	16 MB, shared, static NUCA [22] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories, no silent drops
NoC	4×4 mesh, 256-bit links, X-Y routing, 3 cycles/hop
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (4096 total), 16 commit queue entries/core (1024 total)
***Swarm* instrs**	5 cycles per `enqueue/dequeue/finish_task`
Conflicts	2048-bit 8-way Bloom filters, H_3 hash functions [2] Tile checks take 5 cycles (Bloom filters) + 1 cycle for every timestamp compared in the commit queue
Commits	Tiles and GVT arbiter send updates every 200 cycles

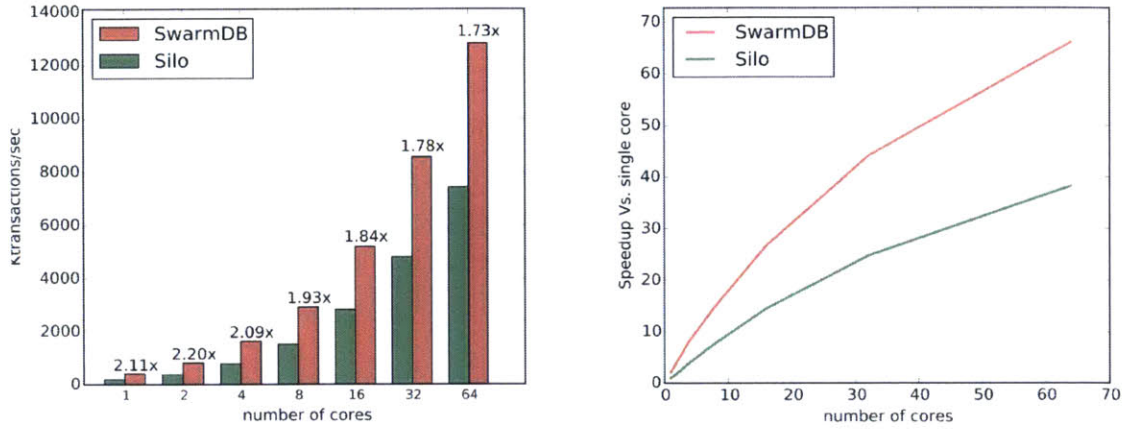
Table 5.1: Configuration of the simulated 64-core CMP (Figure 3-1).

5.2.2 Transaction configuration

Due to time constraint, we did not implement all types of transactions in the TPC-C and TPC-E benchmarks. For TPC-C, we implement *new order* and *payment* transactions, which are usually separately studied in OLTP studies. The transaction mix for TPC-C is 50% *new order* and 50% *payment*. For TPC-E, we implemented *customer position*, *market watch*, *trade status*, *trade order* and *trade update* altogether five types of transactions, which contribute to more than 70% of the transactions TPC-E includes. The transaction mix for TPC-E is: *customer position* (25%), *market watch* (35%), *trade status* (15%), *trade order* (15%) and *trade update* (10%). The ratio of each transaction is chosen according to the specification [40].

In TPC-C, we evaluate *SwarmDB* on two sets of experiments: strong scalability test (where data size remains while core number increases) and weak scalability test (where data size increases along with the core number). For the strong scalability test, we fix the number of warehouses to 4, and scale cores and memory. For the weak scalability test, the number of warehouses matches the number of cores, in which case Silo partitions transactions by warehouse, so each core handles a warehouse.

In TPC-E, we evaluate the performance on the transaction mix mentioned above.



(a) Performance comparison

(b) Speedup comparison

Figure 5-1: Bidding benchmark.

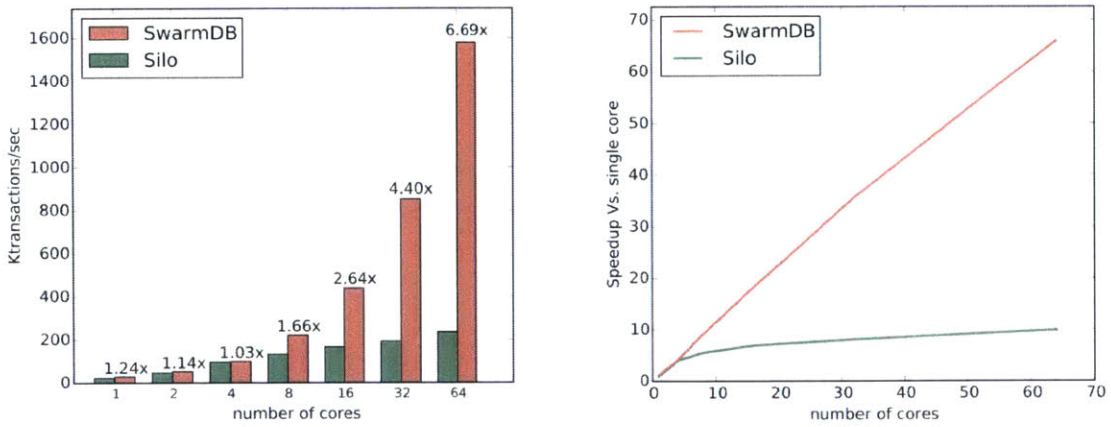
Among these transactions, trade-order and trade-result transactions are write-heavy, with high possibility of having conflict, especially for trade-order transaction, which gets the largest $trade\_id$, and inserts a record with key $trade\_id + 1$ into the trade table. For other transactions, they are read-only.

5.2.3 Performance comparison

We evaluate the system performance by looking at its throughput: thousands of transactions per second (ktransactions/s). To compare the performance of *SwarmDB* and Silo in the scaling systems, we use the throughput of Silo in 1-core system as baseline, and comparing the speedups of *SwarmDB* and Silo under the same system configurations.

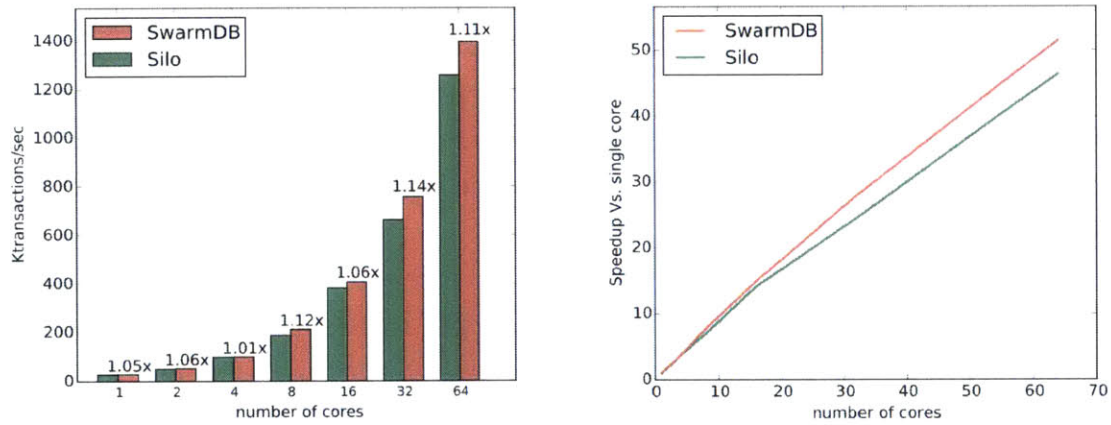
5.3 Performance

Figure 5-1 shows the performance comparison on bidding benchmark. *SwarmDB* outperforms Silo in all cases, even if the benchmark doesn't present the situation where *SwarmDB* could benefit from. The speedup comes from *SwarmDB*'s hardware conflict management comparing to relevant software scheme. Silo has played many performance tricks, for example, using scratch memory reserved on each core for all transactions running on that core. Since *SwarmDB* breaks the transaction and sends queries within one transaction to different cores, this kind of tricks can no longer be applied which will result in the lost of performance. For short transactions, the concurrency control management of Silo outweighs the benefit it gains from its



(a) Performance comparison (b) Speedup comparison

Figure 5-2: TPC-C benchmark with 4 warehouses.

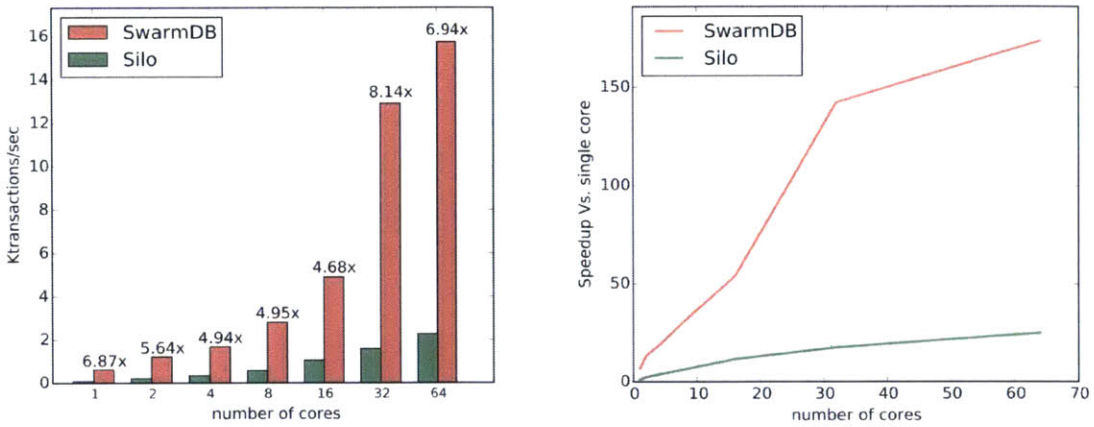


(a) Performance comparison (b) Speedup comparison

Figure 5-3: TPC-C benchmark with scaling warehouses.

performance tricks. However, as the number of cores grows, *SwarmDB* scales not as good as Silo. Since transactions are short and don't usually conflict with each other, the overhead of task management may cause the performance loss. But in all cases, *SwarmDB* is still faster. On this benchmark, *SwarmDB* has average speedup of 1.95x against Silo.

Figure 5-3 and Figure 5-2 shows the performance comparison on TPC-C benchmark. When data size scales with the database size, Silo scales linearly while *SwarmDB* also scales linearly. Since transactions in TPC-C have many queries, dynamically creating memory space for each query is expensive, which compensates the performance gain of eliminating software concurrency control.



(a) Performance comparison

(b) Speedup comparison

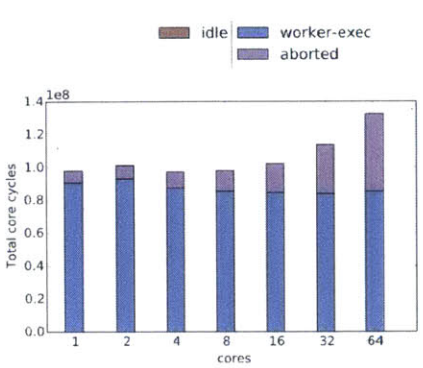
Figure 5-4: TPC-E benchmark.

When data size is fixed and on-chip resources grows, *SwarmDB* significantly outperforms Silo. Silo suffers from high abort rate and large amount of work redoing aborted transaction when the number of cores increases, and scales poorly. *SwarmDB*, however, scales linearly in this case. On a 64-core system *SwarmDB* achieves 6.4x speedup over Silo on the same system. section 5.3 will give detailed analysis of *SwarmDB* performance.

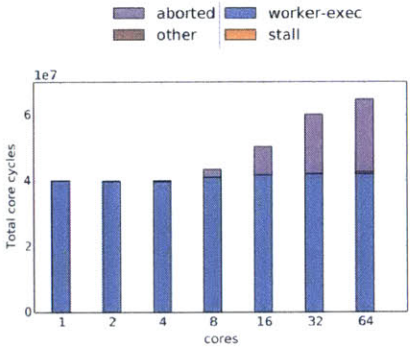
For TPC-E benchmark, *SwarmDB* shows better performance under all system configurations. Since most transactions are read-only, there is little data conflict which can cause the transactions to abort. In this case, *SwarmDB*'s higher throughput comes from more efficient conflict management scheme. Since the transactions are long and scan-heavy, keeping track of all the read/write set in software has significantly higher overhead comparing to doing it in hardware. On a 64-core system, *SwarmDB* outperforms Silo by 6.94x.

5.4 Analysis

Cycle breakdowns: Each set of bars shows results for a single application as the system scales from 1 to 64 cores. Each bar's height reports the cycles spent by each core, normalized by the cycles of the 1-core system (lower is better). With linear scaling, all bars would have a height of 1.0; higher and lower bars indicate sub- and super- linear scaling, respectively. Each bar shows the cycles cores spend executing tasks that are ultimately committed, tasks that are later aborted, stalled and idle (other).

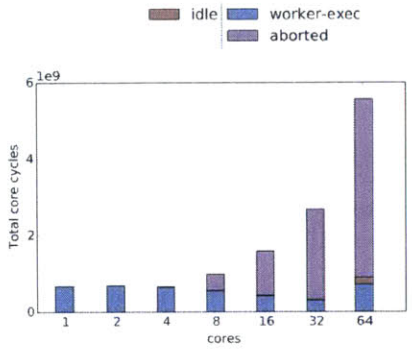


(a) Silo Cycle breakdown: bidding



(b) *SwarmDB* Cycle breakdown: bidding

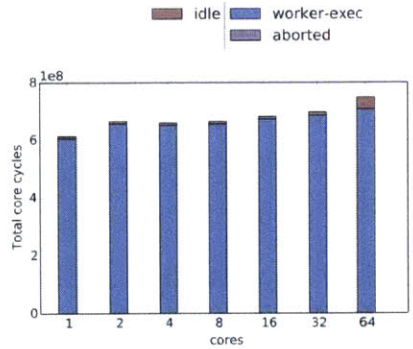
Figure 5-5: Cycle breakdown of Bidding benchmark.



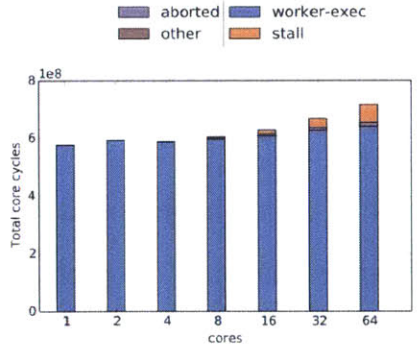
(a) Silo Cycle breakdown: 4 warehouses

(b) *SwarmDB* Cycle breakdown: 4 warehouses

Figure 5-6: Cycle breakdown of TPC-C with 4 warehouses.

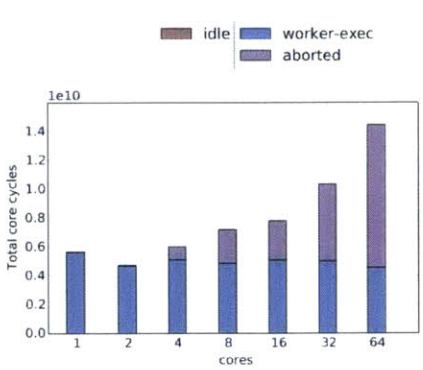


(a) Silo Cycle breakdown: scaling warehouses

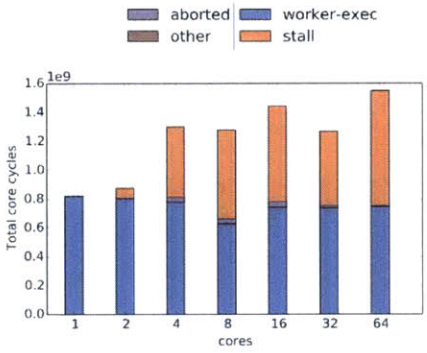


(b) *SwarmDB* Cycle breakdown: scaling warehouses

Figure 5-7: Cycle breakdown of TPC-C with scaling warehouses.



(a) Silo Cycle breakdown: TPC-E



(b) *SwarmDB* Cycle breakdown: TPC-E

Figure 5-8: Cycle breakdown of TPC-E.

For bidding benchmark (Figure 5-5), on a 64-core system, 34.8% of the time is spent on aborted work. Similar time portion is spent on aborted work for Silo. The abort is mostly due to natural data conflict by the benchmark, eg. multiple users placing bid on the same bidding item. Since the transaction in this benchmark is small, so the benefit of exploiting intra-transaction parallelism only improves the performance slightly.

For TPC-C with 4 warehouses (Figure 5-6), most of time is spent on worker thread. Less than 10% of the time is spent on aborted work. For Silo on 4 warehouses, the aborted work contributes to 50% of the total execution time. With fixed warehouses, the workload itself shows heavy contention. However, not all the queries in a single transaction have conflicts. Actually, only the warehouse table and district table is causing conflict, while queries on other tables are mostly safe. *SwarmDB* only aborts the query that is exactly having conflict, instead of aborting the complete transaction.

For TPC-C with scaling warehouses (Figure 5-7), 10% of the time is spent on stalling, which means the system is waiting for tasks. This reveals that *SwarmDB* requires many transactions to be issued at certain times to ensure the system is full loaded, which is a reasonable assumption.

For TPC-E benchmark, the performance gain of *SwarmDB* basically comes from reducing the overhead of concurrency control. With the same number of transactions executed, *SwarmDB* spends only 15% of Silo execution time. However, *SwarmDB*'s performance is hurt by stalling (up to 52%). Because of data partition, the scan and select operation will issue many children tasks to scan/read partitions in parallel, which causes the system to stall on resources like commit queue.

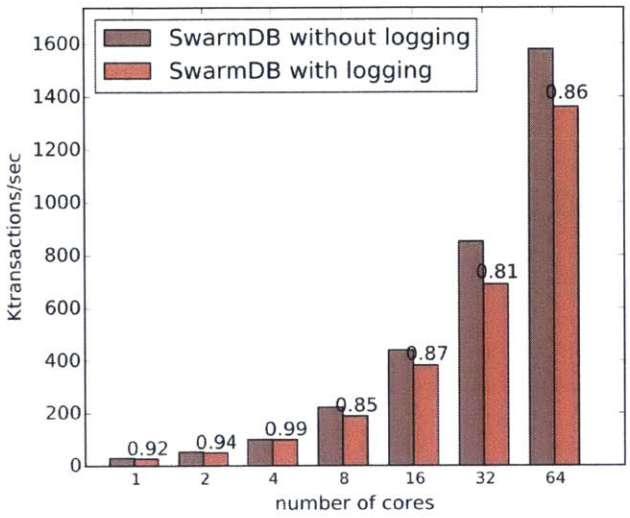


Figure 5-9: Performance with logging.

5.5 *SwarmDB* with Logging

We only tested *SwarmDB* with logging on TPC-C benchmark. Figure 5-9 shows the performance comparison of *SwarmDB* with and without logging. On a 64-core system, the throughput of the database system is only 86% of that without logging, which shows that the logging only introduces very low overhead.

Chapter 6

Conclusion

In this thesis I presented *SwarmDB*, a database system built on the *Swarm* architecture that efficienly exploits fine-grain parallelism in transactional workloads. *Swarm* speculates that most ordering constraints are superfluous, and executes tasks speculatively and out of order but commits them in order. It does efficient hardware conflict detection, tracks the dynamic data dependency among tasks, and implements a distributed, ordered commit scheme. *SwarmDB* leverages these features to achieve high performance. The database needs to be changed in certain way to make full use of the features *Swarm* presents. Breaking down transactions into tasks lets *Swarm* exploit intra-transaction as well as inter-transaction parallelism. It also allows partial aborts by relying on the *Swarm* to track data dependencies among tasks. As a result, *SwarmDB* achieves significant speedup over a state-of-the-art database systems, and shows good scalability on large systems.

Bibliography

[1] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proc. of the 2nd IEEE intl. symp. on High Performance Computer Architecture*, 1996.

[2] J. L. Carter and M. N. Wegman. Universal classes of hash functions (Extended abstract). In *Proc. of the 9th annual ACM Symp. on Theory of Computing*, 1977.

[3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, 2006.

[4] 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 Rec.*, 39(3), 2011.

[5] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Tolerating dependences between large speculative threads via sub-threads. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, 2006.

[6] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. CMP support for large and dependent speculative threads. *Parallel and Distributed Systems, IEEE Transactions on*, 18(8), 2007.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proc. of the 12th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[8] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4), 2013.

[9] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. of the 38th annual Intl. Symp. on Computer Architecture*, 2011.

[10] A. Ferscha and S. K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, University of Maryland, 1998.

[11] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. In *Proc. of the 9th IEEE intl. symp. on High Performance Computer Architecture*, 2003.

[12] S. Ghemawat and P. Menage. TCMalloc: Thread-Caching Malloc http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[13] J. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles, C. Young, et al. Hardware support for fine-grained event-driven computation in Anton 2. In *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[14] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the 8th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture*, 2004.

[16] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[17] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2011.

[18] M. A. Hassaan, D. D. Nguyen, and K. K. Pingali. Kinetic Dependence Graphs. In *Proc. of the 20th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2015.

[19] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, (7), 2008.

[20] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3), 1985.

[21] M. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A Scalable Architecture for Ordered Irregular Parallelism. *Under submission*, 2015.

[22] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[23] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proc. of the 34th annual Intl. Symp. on Computer Architecture*, 2007.

[24] Y. Lev and J.-W. Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-effort Hardware Transactional Memory. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2008.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005.

[26] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture*, 2006.

[27] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *Proc. of the 12th IEEE intl. symp. on High Performance Computer Architecture*, 2006.

[28] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proc. of the 12th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[29] R. Panigrahy and S. Sharma. Sorting and searching using ternary CAMs. *IEEE Micro*, 23(1), 2003.

[30] L. Porter, B. Choi, and D. M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *Proc. of the 18th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2009.

[31] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-level speculation on a CMP can be energy efficient. In *Proc. of the Intl. Conf. on Supercomputing*, 2005.

[32] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *Proc. of the Intl. Conf. on Supercomputing*, 2005.

[33] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[34] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proc. of the 5th International Symposium on Memory Management*, 2006.

[35] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995.

[36] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000.

[37] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th IEEE intl. symp. on High Performance Computer Architecture*, 1998.

[38] T. Strandell. Open Source Database Systems: Systems study, Performance and Scalability. Master's thesis, 2003.

[39] P. Tozun, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, the Ubiquitous, the Unexplored. In *Proc. of the 16th intl. conf. on Extending Database Technology*, 2013.

[40] Transaction Processing Performance Council. TPC-E Benchmark Revision 1.14.0. Technical report, 2015.

[41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proc. of the 24th Symp. on Operating System Principles*, 2013.

[42] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proc. of the EuroSys Conf.*, 2014.

[43] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th IEEE intl. symp. on High Performance Computer Architecture*, 2007.

[44] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proc. of the 11th USENIX symp. on Operating Systems Design and Implementation*, 2014.