Nexion: Enabling Concurrency on Architectures for Ordered Parallelism

by

Robert Benjamin Durfee

S.B. Computer Science and Engineering
Massachusetts Institute of Technology, 2021

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
Daniel Sanchez
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Nexion: Enabling Concurrency on Architectures for Ordered Parallelism

by

Robert Benjamin Durfee

Submitted to the Department of
Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Achieving high performance on modern systems with many cores requires highly parallel applications. Conventional parallel systems require structuring applications into activities that are concurrent, i.e., that may interleave arbitrarily. Concurrency makes it easy for hardware to run these tasks in parallel. However, for most applications, concurrency is challenging to reason about and incurs costly synchronization overheads. To address this problem, recent work has proposed architectures that exploit ordered parallelism. These systems enforce a fixed, programmer-specified order among tasks, and execute tasks speculatively to extract parallelism. Ordered semantics enable parallelism without concurrency, avoiding its complexity, and it is a natural fit for many applications. However, concurrency is also a good fit for many applications, and establishing an order is unnatural and unnecessarily limits parallelism.

We present Nexion, an execution model that supports concurrency alongside ordered parallelism. Programmers split applications into short tasks that can be given timestamps to specify order constraints. Groups of tasks can be marked as concurrent and will execute independently if no data is shared among them. If data is shared, Nexion ensures that tasks remain atomic and respect applicable timestamp orders. We extend Swarm, an architecture for ordered parallelism, with minimal additional state to implement Nexion. The implementation is distributed and only involves communication between tasks that share data. On evaluated benchmarks, Nexion improves overall scalability by up to 32× over software-only solutions and up to 2.4× over the Swarm baseline architecture.

Thesis Supervisor: Daniel Sanchez
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis would not be possible without the support of several individuals, to whom I offer my deepest thanks.

First and foremost, I would like to thank my advisor, Daniel Sanchez. If not for his encouragement from the beginning, I would not have pursued this project. Whenever I was skeptical of the significance of my ideas, he provided the reassurance necessary to keep me engaged. At numerous points along this journey, I was convinced I had hit an insurmountable roadblock. But after each of our meetings, not only did he unblock my progress, he provided multiple promising avenues to explore. His depth and breadth of knowledge, not only in computer architecture, but across many related (and unrelated) fields continues to astound. Though perhaps more impressive is his ability to effortlessly explain the most complex technical topics in the simplest, most understandable language. I am very lucky to have had him as an advisor.

I also had the great pleasure of working frequently under the direct mentorship of Victor Ying. It was Victor's work that originally drew my attention to the Sanchez group. While I was an undergraduate student, he introduced me to research, guiding me on my first project. He continuously encouraged me to pursue my own interests, and was always excited to discuss new and interesting ideas. When I became a graduate student, he graciously continued to lend his time to brainstorm ideas and pair-debug complex deadlock and livelock scenarios. In fact, many of the core ideas of this thesis originate from our brainstorm sessions. In his absence during the last month of my project, I was reminded just how crucial his support was to this work whenever I found myself heading to his office to discuss an issue.

I am also thankful to all the members of the Sanchez group: Joel Emer, Nithya Attaluri, Fares Elsabbagh, Axel Feldmann, Kendall Garner, Aleksandar Krastev, Ryan Lee, Quan Nguyen, Nikola Samardzic, Shabnam Sheikhh, and Yifan Yang. In addition to providing valuable feedback on my presentations, they have all become my close friends. I have thoroughly enjoyed all of our lively technical and non-technical discussions prompted by the now-immortal, "Any news?". Thanks specifically to Fares

for being the first reviewer of this thesis. His comments were invaluable as they helped focus my efforts on the most important aspects in preparation for future revisions. I also thank Quan for his consistent words of encouragement throughout my time in the group, and particularly in the last month of my project. Whenever I had a question, he never turned me away, even when he was in the final hours before his Ph.D. thesis deadline.

Last but not least, I would like to thank my family. My sister, Amanda, has always been my most influential role model. If not for her, I would not have made it to MIT. She consistently challenges me to better myself, for which I am immeasurably grateful. And finally, my parents, Ben and Amy, who raised me in a stimulating home which shaped me into the person I am today. Throughout my time at MIT, they have shown truly unconditional love and support. Without their love, guidance, and insight, this thesis would not exist.

Contents

Chapter 1

Introduction

To use modern systems well, applications must be highly *parallel*, i.e., be structured into many tasks that can run at the same time. Parallelism is crucial because current fabrication technology has enabled chips with many thousands of functional units. Over time, computer systems have moved from exploiting implicit parallelism in sequential programs to requiring programs to be explicitly parallelized.

Until 2005, out-of-order cores uncovered enough instruction-level parallelism to improve performance without programmer involvement. However, this approach has reached its limits [2, 9, 16]. To continue providing performance improvements, architectures switched from single-core to multicore designs. In these systems, sequences of instructions are divided into individual threads. These threads are mapped to cores to be executed in parallel. Threads execute independently and thus they are *concurrent*, i.e., they may interleave arbitrarily.

When applications have abundant concurrency, it is easy to exploit parallelism. Unfortunately, reasoning about concurrency is challenging and comes with costly overheads. Since programs operate on shared data, some order constraints between threads must be enforced. To enforce order, programmers must use explicit synchronization primitives such as locks and barriers. This explicit synchronization is very error-prone and adds overheads, limiting application scalability.

To avoid the pitfalls of concurrency, the recent Swarm architecture [18,29] explores a complementary approach that seeks to provide high scalability while maintaining

order. Swarm enables high performance by leveraging *ordered parallelism* [15, 23], where an application is split into ordered tasks that appear to execute in a program-defined order. To uncover parallelism, Swarm executes these tasks speculatively and out of order. By exposing timestamps to programs, Swarm is able to parallelize more algorithms than other ordered speculation techniques like thread-level speculation (TLS) [10,12,13,27,28]. However, for some applications, it is more natural to uncover parallelism using available concurrency. Requiring an order for all tasks limits this parallelism and makes programming unnecessarily cumbersome.

In summary, uncovering parallelism is crucial to enable high performance. Available concurrency makes this easy to exploit, but it is challenging to reason about in many cases. Relying on order semantics lessens this burden. Nevertheless, some applications naturally have concurrency, so it is better to exploit it if it exists. As a result, systems should support both ordered parallelism and concurrency. Unfortunately, prior systems have failed to emphasize the benefits of supporting both, instead focusing on one at the expense of the other.

The goal of this work is to easily support the expression and exploitation of concurrency of independent activities without sacrificing support for ordered parallelism. Our first contribution is *Nexion*, an execution model that supports concurrency and ordered parallelism (Section 3). In this execution model, a program consists of tasks belonging to disjoint *realms*. Realms have ordered semantics similar to Swarm, and concurrency is supported across realms. In particular, tasks within a realm appear to execute atomically in timestamp order, although they may be executed speculatively out of order. If no data is shared, tasks across realms will execute and commit completely independently from each other. When data is shared among realms, Nexion ensures that tasks still execute atomically. Furthermore, tasks sharing data follow a serializable order that respects the timestamp order of each realm.

Our second contribution is an implementation of Nexion that builds on Swarm (Section 4). This implementation supports concurrency alongside ordered parallelism with less than 16KB of additional state in a 256-core system. Like in Swarm, Nexion executes tasks speculatively out of order and forwards speculative data, even among

realms. A distributed mechanism maintains a serializable order of tasks across realms, selectively aborting only the tasks involved in an order violation. This mechanism involves communication only among realms in which data is shared and builds on the observation that speculative data is rarely forwarded across realms.

We evaluate Nexion on a state-of-the-art, in-memory transactional database (Section 5). Nexion demonstrates increased scalability over prior systems. In particular, Nexion demonstrates up to 32× the scalability of software-only benchmarks and up to 2.4× the scalability of the Swarm baseline on 256 cores.

Chapter 2

Background and Motivation

We motivate Nexion using the case study of a transactional database. Consider a database that consists a handful of transactions interacting with multiple tables. Each transaction involves several queries. Parallelism exists both within and among transactions. The total work per transaction is variable, not known a priori, and often highly skewed (i.e., many transactions access the same subset of tuples, leading to many conflicts).

This case study explores parallelizing a transactional database using three different classes of systems: (1) a software-only approach using a commodity multicore; (2) using atomic transactions offered by HTM systems; and (3) leveraging recent architectural support for ordered parallelism using Swarm.

2.1 Conventional Multithreading

In commodity multicore systems in widespread use today, the classic programming model presents very little abstraction over the hardware. To utilize multiple cores in an application, a programmer can create multiple threads, which are mapped onto cores by the operating system. To parallelize a transactional database, each thread is typically given a set of transactions to execute, and runs them one by one. In this implementation, independent transactions can remain unordered among threads and execute concurrently.

However, since these transactions may operate on shared data, but must appear to execute atomically, some form of synchronization is required. This problem has a rich history of prior work that has establishe a whole taxonomy of solutions. At the broadest level, these solutions are split into two primary categories: pessimistic and optimistic [4]. Two-phase locking (2PL) is the main implementation strategy for pessimistic concurrency control. With 2PL, transactions have to acquire locks for a particular element in the database before they are allowed to execute a read or write operation on that element [5, 11]. Optimistic concurrency control, on the other hand, speculates conflicts will not occur, and must recover upon miss-speculation. To achieve this, implementations often assign each transaction with a unique, monotonically increasing timestamp prior to execution, resolving conflicts by comparing timestamps [4,20]. Prior work has shown that there are bottlenecks in both categories, preventing implementations from scaling to high core counts [33]. Furthermore, these techniques only exploit parallelism across transactions, not within each transaction.

2.2 Hardware Transactional Memory

To alleviate some of the challenges of software-based concurrency control mechanisms discussed in Section 2.1, hardware transactional memory (HTM) systems can be used to provide hardware support for atomicity. With HTM, sequences of instructions can be wrapped together into transactions. The HTM system guarantees that each transaction will execute atomically with respect to other concurrent transactions. A transactional database can easily utilize this hardware support by mapping each database transaction to an HTM transaction. This abstracts away the challenges of implementing parallel databases in software while providing the same guarantees.

The semantics of prior HTM systems focus on providing atomicity and isolation guarantees. The differences in implementations vary primarily in the way they provide these guarantees through version management and conflict detection. To manage versions of speculative data, the HTM system can either employ eager or lazy strategies. Eager version management implementations put the new value in place [3,21,32]

and lazy implementations temporarily leave the old value in place [3, 14, 24]. When the data written by a transaction intersects the data read or written by another concurrent transaction, this signals a conflict. HTM systems can detect conflicts eagerly or lazily. Implementations detect conflicts eagerly if offending loads or stores are identified immediately [3, 21, 24, 32] and lazily if this detection is deferred to a later time [14].

These implementations enable transactions that do not share data to execute concurrently, only serializing when data is shared. However, most HTM systems do not provide ordering guarantees [17, 21] and those that do [14] provide simple barrier semantics which are unfit for exploiting ordered parallelism within transactions.

2.3 Architectures for Ordered Parallelism

Up to this point, the parallel implementations of a transactional database have only succeeded in exploiting the available concurrency across transactions. If this application is to scale beyond a few dozen cores, all available parallelism must be extracted. To extract the parallelism within a transaction, we consider the recent Swarm architecture and its extensions.

2.3.1 Swarm

Swarm provides a rich execution model that can scalably exploit ordered parallelism [18]. A Swarm program consists of tasks with programmer-specified timestamps. A task can create children tasks with a timestamp equal or greater than its own. Swarm enforces that all tasks appear to execute atomically and in timestamp order. If two tasks have the same timestamp, the order of execution is arbitrary.

To implement a transactional database with Swarm, each transaction is broken down into several small tasks as shown abstractly in Figure 2-1. If these tasks in each transaction have necessary order constraints, these are reflected in the timestamp order. However, Swarm does not provide atomicity guarantees across groups of tasks. As a result, as shown in Figure 2-1, all tasks within each transaction must have

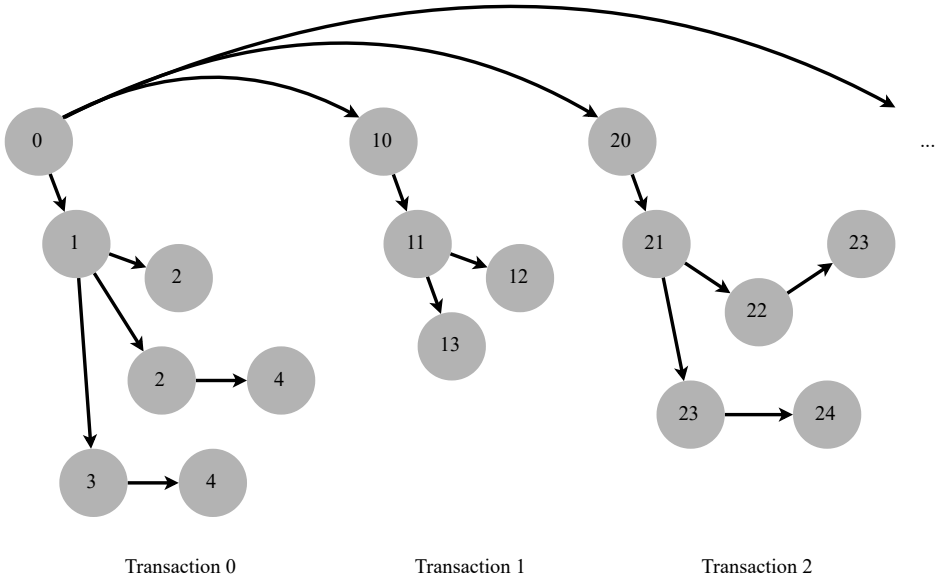


Figure 2-1: Database transactions parallelized with Swarm.

disjoint timestamps from other transactions, otherwise the atomicity requirements of the transactional database will not be satisfied.

This removes support for concurrency as all transactions are given a sequential order, even if they do not operate on shared data. This especially causes issues when tasks have large differences in work. Suppose that in Figure 2-1 task 3 takes thousands of cycles, while later tasks take only tens of cycles. The only task that directly depends on this task is its child, task 4. However, because of the established order, every later-timestamped will not be allowed to commit until task 3 commits, even if they have no data dependencies.

Furthermore, this is highly cumbersome for programmers, as each each transaction has a variable amount of work, so it is challenging to ensure transaction timestamps do not overlap. For example, in Figure 2-1, if transaction 1 at some point spawns a child tasks with timestamp 20 or higher, this could make both transactions 1 and 2 no longer atomic. As a result, Swarm effectively harnesses all parallelism available within transactions, but fails to exploit any concurrency across unrelated transactions and has a restrictive programming model.

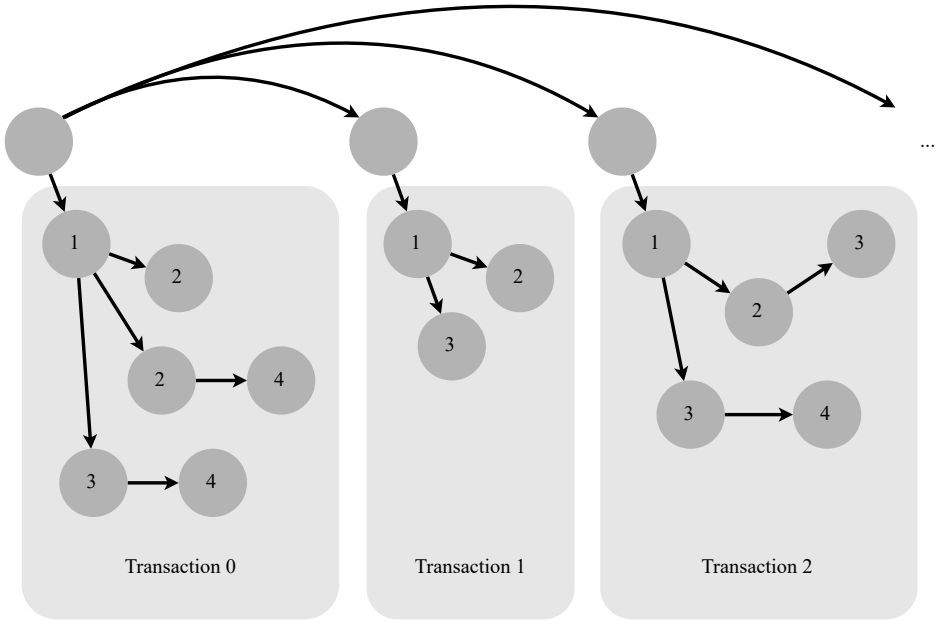


Figure 2-2: Database transactions parallelized with Fractal.

2.3.2 Fractal

The Fractal extension to Swarm provides the same semantics as Swarm with the ability to group tasks into domains [29]. Furthermore, these domains can be either ordered or unordered. Now, the implementation from Figure 2-1 can be simplified as shown in Figure 2-2. Each transaction consists of a single domain of ordered tasks and each domain is unordered with respect to other domains. Fractal appears to solve all issues by exploiting all ordered parallelism within transactions and available concurrency across transactions.

However, behind the scenes, upon dequeue of the first task in each domain, Fractal assigns an order to each unordered domain. As a result, even though the domains are unordered from the viewpoint of the programmer, they are ordered in the implementation. Thus Fractal improves only slightly over Swarm as independent transactions with no shared data are still ordered, but now this isn't specified by the programmer, happening instead on the first task's dequeue.

2.4 Takeaways

In summary, conventional multithreading techniques to scale a database can exploit concurrency across transactions, but not ordered parallelism within transactions. Furthermore, this is only possible using sophisticated techniques, which are still limited by various bottlenecks. HTM support can alleviate some of the implementation complexities and scalability issues, but the semantics are insufficient to harness inter-transaction parallelism. Architectures for ordered parallelism can exploit this parallelism within transactions, but not without losing support for concurrency. Nexion addresses these issues by providing hardware support to effectively exploit concurrency across transactions and parallelism within transactions.

Chapter 3

Nexion Execution Model

Figure 3-1 depicts the key elements of the Nexion execution model. Nexion programs consist of tasks located in realms. Realms are configured prior to program execution. Each task belongs to a single realm and can enqueue additional children into its same realm. For example, in Figure 3-1, realm 0 has two tasks, U and V, with timestamps 0 and 4, respectively.

3.1 Semantics Within a Realm

Within a realm, semantics are similar to those of Swarm. Each task has a program-specified timestamp. A task can enqueue child tasks into the same realm with any timestamp equal to or greater than its own. Nexion guarantees that tasks appear to run in increasing timestamp order. If multiple tasks have the same timestamp, Nexion arbitrarily chooses an order among them. This order always respects parent-child dependencies. Timestamps let programs convey their specific order requirements. For example, in Figure 3-1, realm 2's tasks Y and Z have timestamps 3 and 32, respectively, to reflect the requirement that task Y must write to C before task Z reads from C.

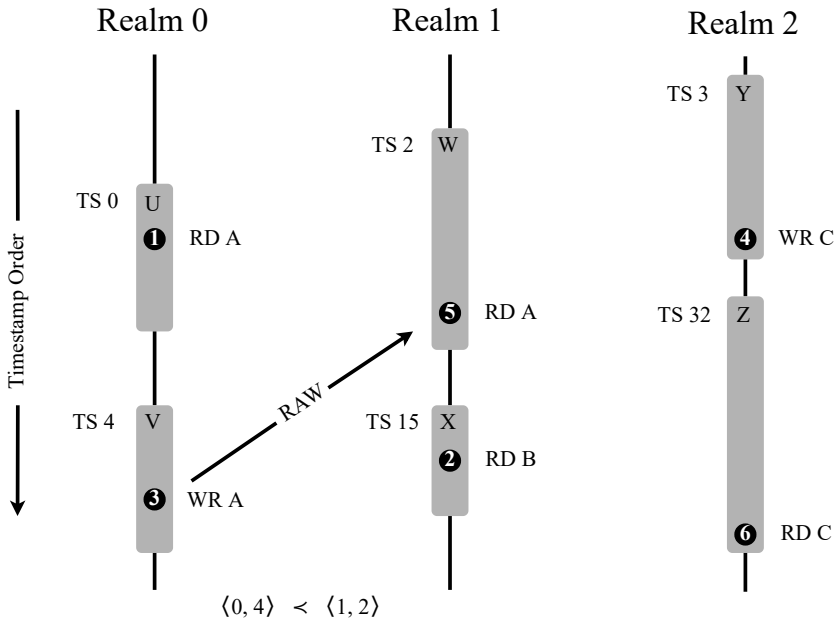


Figure 3-1: Elements of the Nexion execution model.

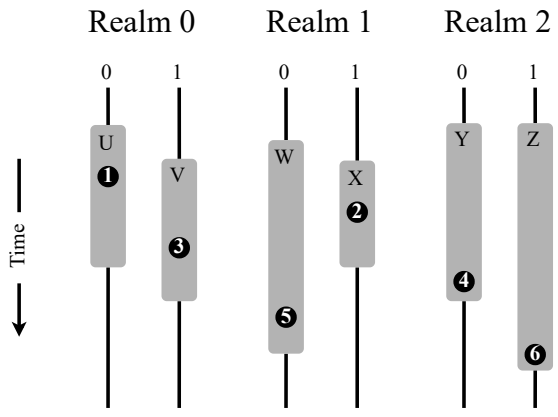


Figure 3-2: The tasks in Figure 3-1 are executing speculatively in parallel.

3.2 Semantics Across Realms

Unlike tasks within a realm, tasks across realms do not have a predefined program order. If two realms never access the same data, they can execute concurrently. For example, in Figure 3-1, realm 1 and realm 2 never access the same data. Realm 1 only accesses A and B, while realm 2 only accesses C. As a result, realms 1 and 2 can execute concurrently.

If two realms access the same data, Nexion guarantees that tasks execute atomically and in a serializable order that respects the timestamp order in both realms. For example, in Figure 3-1, realms 0 and 1 are initially independent up to when task W reads A. Previously, task V from realm 0 wrote to A. This read-after-write (RAW) data dependency establishes an ordering between realms 0 and 1. Nexion will guarantee that task V (and all tasks with timestamps earlier than 4 in realm 0) appears to happen atomically before task W (and all tasks with timestamps later than 2 in realm 1).

3.3 Programming Interface

Nexion's low-level programming interface is very similar to the interface of Swarm. In Nexion, tasks are simply functions with signature:

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

Prior to the execution of a Nexion program, the programmer must enqueue at least one initial task into each realm by calling:

```
enqueueInit(realm, taskFn, timestamp, args...)
```

This establishes the realms that will be present for the entire duration of the program. After a Nexion program begins execution, code can enqueue other tasks into the same realm by calling:

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

Enqueuing tasks into different realms during program execution is not supported.

Chapter 4

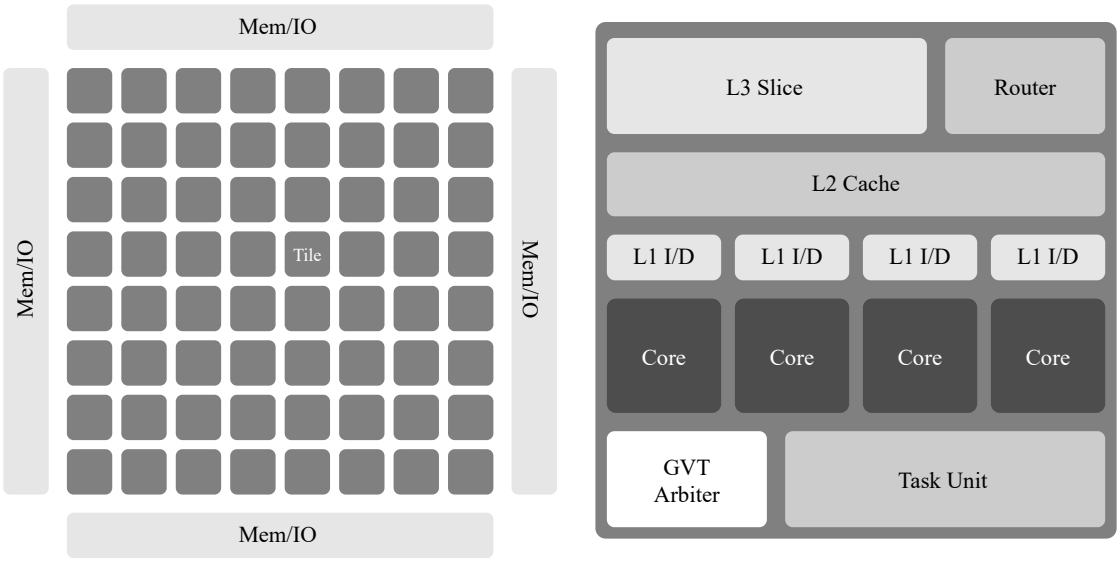
Nexion Implementation

We present an implementation of Nexion targeting three primary design goals. First, if tasks across realms are completely independent, they should proceed concurrently and not incur any serialization overheads. Second, existing Swarm semantics should be preserved across realms whenever possible. Finally, tasks should be able to access arbitrary data, including data across realms, but access to local data should remain cheap as it is the common case.

We present Nexion in a layered fashion. First, we describe the important aspects of the Swarm baseline microarchitecture. Then, we present the modifications necessary for Nexion, starting with an idealized implementation and working toward a final, realistic hardware implementation.

4.1 Baseline Swarm

Nexion builds on Swarm, which introduces modest changes to a tiled, cache-coherent multicore. Figure 4-1 shows the baseline Swarm design. Each tile has a group of simple cores, each with its own private L1 cache. All cores in a tile share an L2 cache, and each tile has a slice of a fully-shared L3 cache. Every tile is augmented with a task unit and a global virtual time (GVT) arbiter. The task unit queues, dispatches, and commits tasks. The GVT arbiter communicates with other GVT arbiters to enable scalable commits.



(a) Swarm chip.

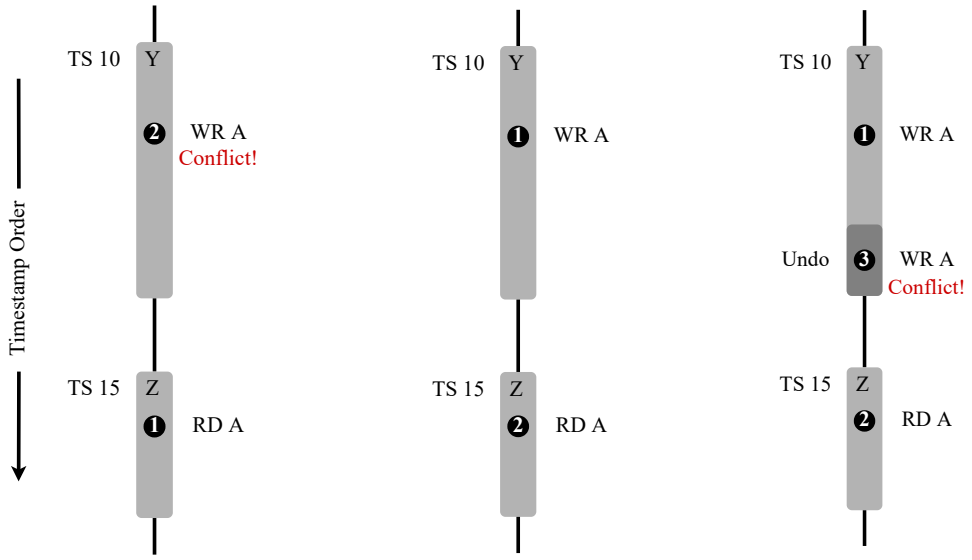
(b) Swarm tile.

Figure 4-1: 256-core Swarm chip and tile configuration.

Swarm uncovers parallelism by executing tasks speculatively and out of order. To uncover enough parallelism, Swarm can speculate thousands of tasks ahead of the earliest unfinished task by utilizing large task queues. Swarm uses eager versioning to store speculative data in place and logs old values to make commits fast and enable simple forwarding of speculative values. Swarm hardware efficiently and scalably maintains program order with three main mechanisms: virtual time-based conflict detection, selective aborts of dependent tasks, and distributed ordered commits.

4.1.1 Conflict Detection

In Swarm, conflict detection is based on the program-specified, priority order. Conflicts occur when a task accesses an address that was previously accessed by a later-timestamped task. For example, in Figure 4-2a, tasks Y and Z both access address A. Task Y should appear to execute before task Z because it has an earlier timestamp. However, to uncover maximum parallelism, tasks Y and Z are executing speculatively in parallel. In this case, task Z happens to read A before task Y writes A. This conflicts with the priority order and triggers an abort. On the other hand, in Figure



(a) Task Z accesses A before Task Y, resulting in a conflict.

(b) Task Y accesses A before Task Z, following timestamp order.

(c) Upon abort, Task Y accesses A after Task Z, resulting in a conflict.

Figure 4-2: Two tasks access the same address in different orders in Swarm.

4-2b, task Y writes A before task Z reads A. This respects the priority order and, thanks to eager versioning, task Z will read the most up-to-date version of A, even while it is speculative.

To avoid conflict-checking with all running and finished (but waiting to commit) tasks in the system, Swarm leverages the cache hierarchy to filter checks. If the access hits in the L1 cache, it is guaranteed to be conflict-free. This is accomplished by flushing the L1 when dequeuing a task with an earlier timestamp than the one it just finished. If the access misses in the L1, the access is conflict-checked locally against tasks in the same tile, but may need to check tasks in other tiles as well. If the access hits in the L2, the associated cache line maintains a *canary timestamp*, which is the lowest timestamp for which a global conflict check is unnecessary. If the access has a lower timestamp than the canary, or the access misses in the L2 cache, it is conflict-checked against tasks in any tile which might share the conflicting cache line (as determined using sticky bits [21]). To efficiently maintain and check for read/write set intersection between tasks, Swarm uses hardware Bloom filters [32].

4.1.2 Selective Aborts

Once a conflict has been established, Swarm aborts the later-timestamped task and all of its dependents. The task's dependents consist of its children and all tasks that accessed its speculatively written data. The abort process consists of three steps. First, the task's children are notified of abort. Second, the task's undo log is walked in last-in, first-out order, restoring previous values. If these undo writes trigger conflicts, the task waits for the aborts to complete before continuing. Third, the task's read/write sets are cleared and it is removed from the task queues. This strategy is applied recursively, aborting all dependent tasks.

The key enabling element of this abort strategy is the reuse of the conflict detection mechanism. The writes during the undo phase trigger conflicts just as normal writes. Consider Figure 4-2c where task Y been notified to abort. Just as before, task Z has read speculative data written by task Y and should also abort. When walking its undo log, task Y will write again to address A. This write will conflict with task Z's read to A and correctly trigger an abort of task Z. With large speculation windows, explicit tracking of dependencies would be prohibitively expensive. This strategy avoids this problem by using the conflict detection mechanism to recover dependencies as needed.

4.1.3 Scalable Ordered Commits

To enable high-throughput, scalable commits that respect the program-specified order, Swarm modifies the virtual time algorithm. In the original Swarm paper, each tile will periodically (e.g., every 200 cycles) send the minimum timestamp of unfinished tasks in that tile to an arbiter. The arbiter will compute the minimum timestamp of all unfinished tasks, called the global virtual time (GVT). The GVT is then broadcast to all tiles which commit all tasks up to that timestamp. This strategy avoids serializing commits and amortizes the cost of commits across many cycles. As a further optimization, instead of only having one GVT arbiter, current implementations of Swarm achieve this all-to-one reduction and one-to-all broadcast using a tree of GVT arbiters, with one arbiter per tile.

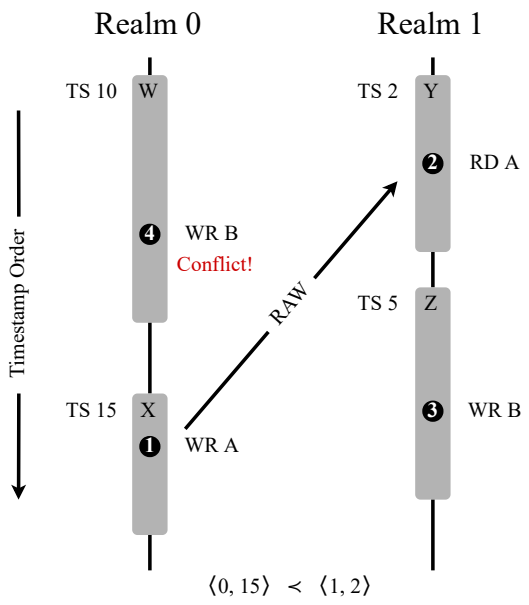


Figure 4-3: Two realms share speculative data, imposing a partial order of timestamps and triggering a conflict.

4.2 Conflicts in Nexion

The definition of a conflict in Swarm from Section 4.1.1 states that a conflict occurs when a task accesses an address that was previously accessed by a later-timestamped task. This definition establishes two necessary conditions for a conflict: tasks must access the same data and the accesses must violate the established order. In Nexion, tasks may belong to disjoint realms. If data is never shared among realms, the first condition is never satisfied and conflicts across realms are impossible. In this case, the implementation of Nexion is simple. Each realm is an independent instance of Swarm that operates in complete isolation from other realms. The conflict definition remains unchanged from Swarm.

However, if data is shared across realms, the first condition may be satisfied. To extend this definition of a conflict while maintaining existing Swarm semantics, Nexion must determine when the second condition is met and a conflict occurs due to an order violation. This requires a partial order of tasks. Tasks within a realm have a partial order specified by the program, but tasks across realms do not have

an established order. Nexion imposes a partial order on the timestamps of different realms, encompassing the program-specified order within each. Given this imposed partial order on all tasks, the Swarm definition of a conflict remains consistent for tasks both within and across realms. That is, a conflict occurs if data is shared across realms, and accesses violate the imposed partial order.

Timestamps of different realms are initially incomparable. Nexion imposes a partial order on timestamps only when two tasks in different realms access the same speculative data and at least one of the accesses is a write. This creates a speculative data dependency in one of three classic forms: read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW). Each speculative data dependency induces a happens-before order on timestamps of different realms. For example, consider Figure 4-3 with two realms. Each realm consists of a two tasks, all four of which are executing speculatively in parallel. First, task X writes to address A. Then, before task X commits, task Y reads A. This establishes a RAW speculative data dependency. Given this dependency, Nexion imposes a happens-before order such that realm 0's timestamp 15 happens before realm 1's timestamp 2 (i.e., $\langle 0, 15\rangle \prec \langle 1, 2\rangle$). Now, this happens-before order must be respected on subsequent accesses.

Later, as shown in Figure 4-3, task Z writes to address B. Then, task W also writes to address B. Even though tasks Z and W are from different realms, this introduces a conflict as Nexion's happens-before order dictates that task Z has a later timestamp than task W. As a result, this will trigger an abort.

Happens-before orders established by Nexion are also transitive. Consider the case in Figure 4-4 with three realms. All tasks are executing speculatively in parallel and have not committed. Tasks W and X establish a RAW dependency on address A imposing $\langle 0, 31\rangle \prec \langle 1, 15\rangle$. Tasks X and Y establish a WAR dependency on address B imposing $\langle 1, 15\rangle \prec \langle 2, 1\rangle$. When task V attempts to write address C after task Z has written to C, this will trigger a conflict because of the transitive partial order $\langle 0, 31\rangle \prec \langle 1, 15\rangle \prec \langle 2, 1\rangle$.

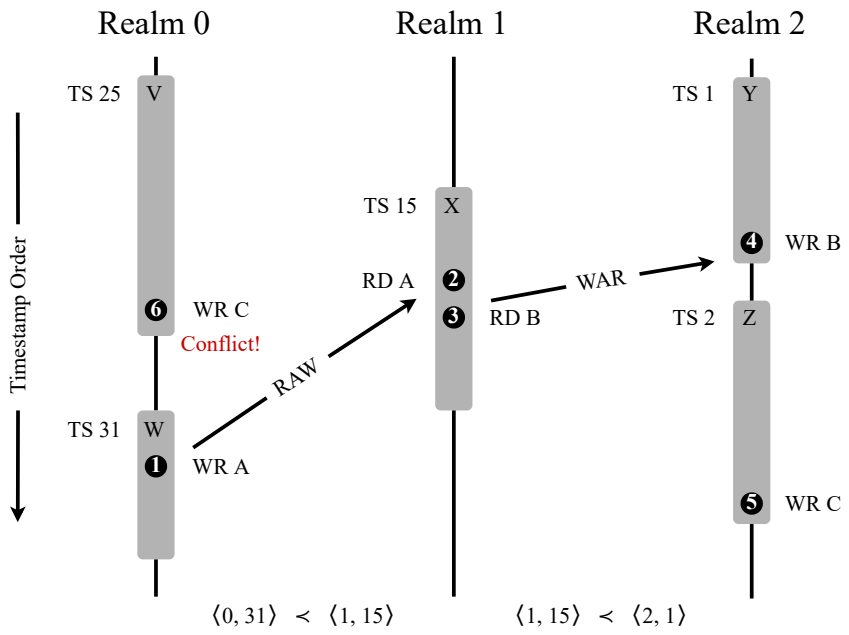
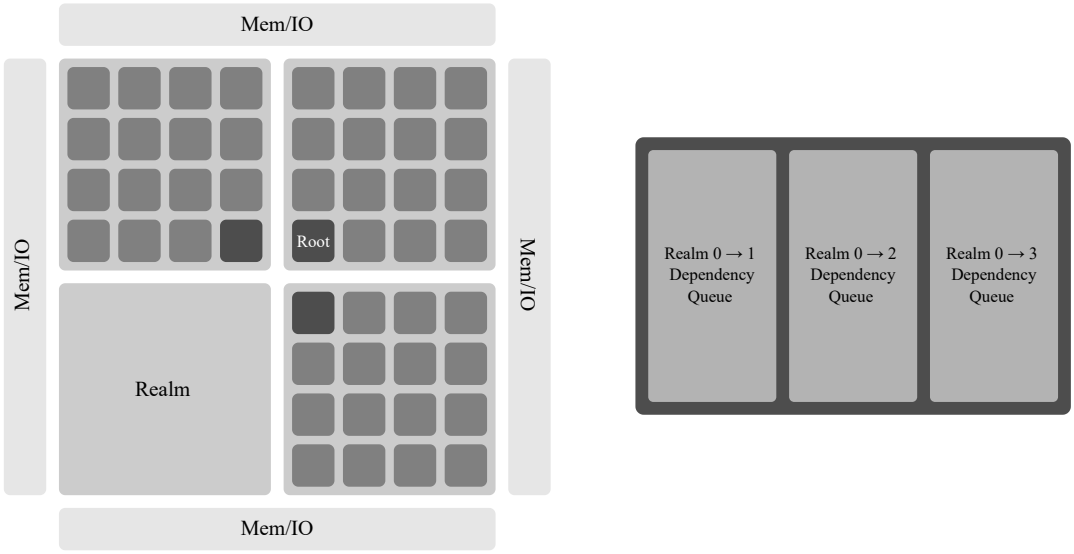


Figure 4-4: Three realms share speculative data, imposing a transitive partial order of timestamps and triggering a conflict.

4.3 Nexion with Infinite Resources

With the extended definition of a conflict in place, we present an implementation of Nexion. The main components of Nexion are pictured in Figure 4-5. In Nexion, realms are established as static partitions of tiles, predetermined in hardware. By partitioning the system by tiles, Nexion is able to leverage the hierarchical memory system to dramatically reduce the cost and complexity of implementation. Each partition is similar to an instance of Swarm. The differences lie in the realm virtual time (RVT) arbiters, which house the state and logic necessary to implement Nexion. Each partition has it's own tree of RVT arbiters mapped to tiles, with the root RVT arbiter in each realm maintaining additional state shown in Figure 4-5b. Before describing a realistic hardware implementation, this section describes the high-level changes necessary to Swarm, assuming unbounded resources.



(a) Nexion chip.

(b) Nexion root RVT arbiter.

Figure 4-5: 256-core, 4-realm Nexion chip and tile configuration.

4.3.1 Conflict Detection

In Swarm, ignoring optimizations, conflict detection is triggered on memory accesses and involves two steps:

1. Identify unfinished tasks that have accessed the current address.

2. Check if any of the accessor tasks' timestamps are later than the current task's timestamp.

In Nexion, if all accessor tasks belong to the current task's realm, then this process remains unchanged. However, if any accessor task belongs to a different realm, this process becomes more complicated as what constitutes a "later" timestamp depends on the imposed happens-before order.

To provide this context, whenever a speculative data dependency between different realms arises without conflict, Nexion records the resulting timestamp happens-before order in a queue. These queues are maintained in the root RVT arbiter in each realm. For example, in Figure 4-4, Nexion would record the happens-before orders $\langle 0, 31\rangle \prec \langle 1, 15\rangle$ and $\langle 1, 15\rangle \prec \langle 2, 1\rangle$.

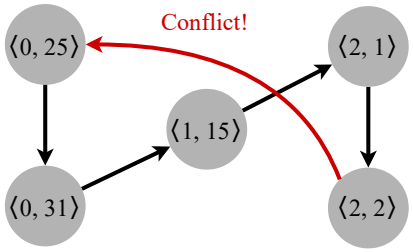


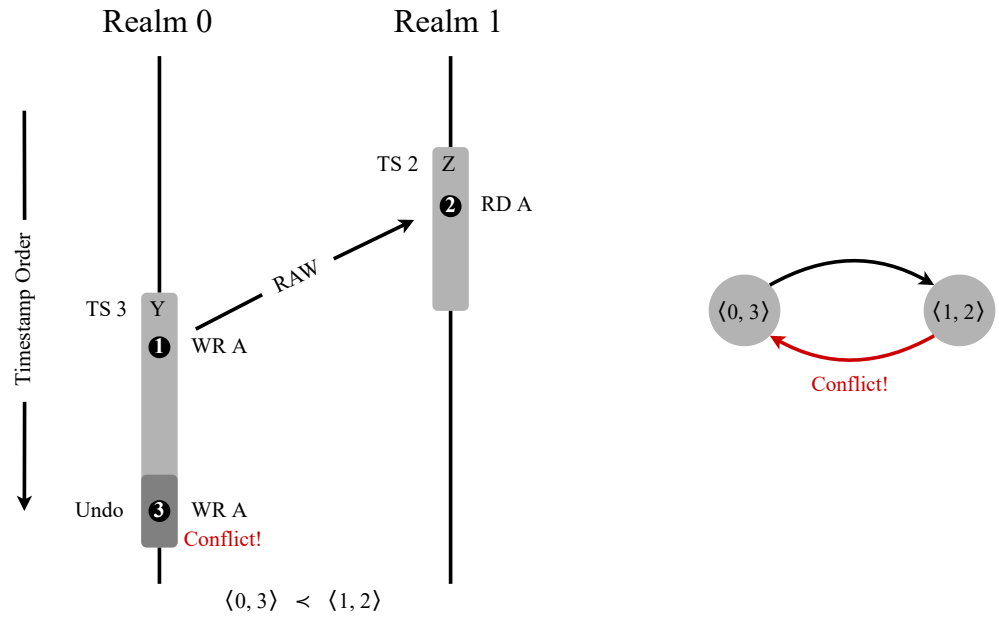
Figure 4-6: Cycle forms in the happens-before graph associated with Figure 4-4. In each node, the first number refers to a task's realm and the second to the task's timestamp.

These happens-before orders, in combination with the implicit timestamp orders within realms, form a directed acyclic graph (DAG) of all timestamps within Nexion. The order DAG for the example in Figure 4-4 is shown in Figure 4-6. To detect conflicts, Nexion uses the happens-before orders in the queues to construct this DAG. Before allowing a new speculative data dependence, the associated happens-before order is presented to the root RVT arbiter and tentatively introduced to the DAG. The graph is traversed using a depth-first search (DFS) to determine if the new order introduces a cycle. If a cycle is introduced, this is a conflict and triggers an abort and the tentative happens-before relation is not recorded.

Prior work prevents cyclic data dependencies using techniques that do not involve explicitly traversing a DAG [17, 21, 25]. However, these mechanisms often cause false cycles and provide weaker semantics than Nexion. The specifics of this DFS traversal and prior work are discussed in Section 4.7.

4.3.2 Selective Aborts

Once a conflict is established, Nexion aborts the later-timestamped task involved in the conflict and all of its dependents, given the imposed happens-before order of timestamps across realms. The same three-step process for aborts from Swarm described in Section 4.1.2 is followed in Nexion. As in Swarm, the key insight for selective aborts in Nexion is the reuse of the conflict detection mechanism: the writes during the undo phase trigger conflicts just as normal writes. Consider Figure 4-7a,



(a) Upon abort, task Y accesses A again after task Z, triggering a selective abort across realms.

(b) Cycle forms in the happens-before graph associated with Figure 4-7a.

Figure 4-7: Selective aborts in Nexion.

where task Y has been notified to abort. Task Z has read speculative data written to address A by task Y and should also abort. In the undo phase, task Y will write again to A. This would introduce a WAR speculative data dependency between the realms. However, since this introduces a cycle in the happens-before graph shown in Figure 4-7b, a conflict occurs. This correctly triggers an abort of task Z because the recorded happens-before order dictates that task Z has a later timestamp than task Y. Applied recursively, this abort strategy ensures that all dependent tasks will be aborted, even across realms.

4.3.3 Scalable Ordered Commits

Similar to Swarm, Nexion uses a modified virtual time algorithm for scalable, high-throughput commits in program order. However, instead of having a single global virtual time (GVT), Nexion maintains multiple realm virtual times (RVTs) to enable concurrent commits across realms. Each RVT is calculated independently using

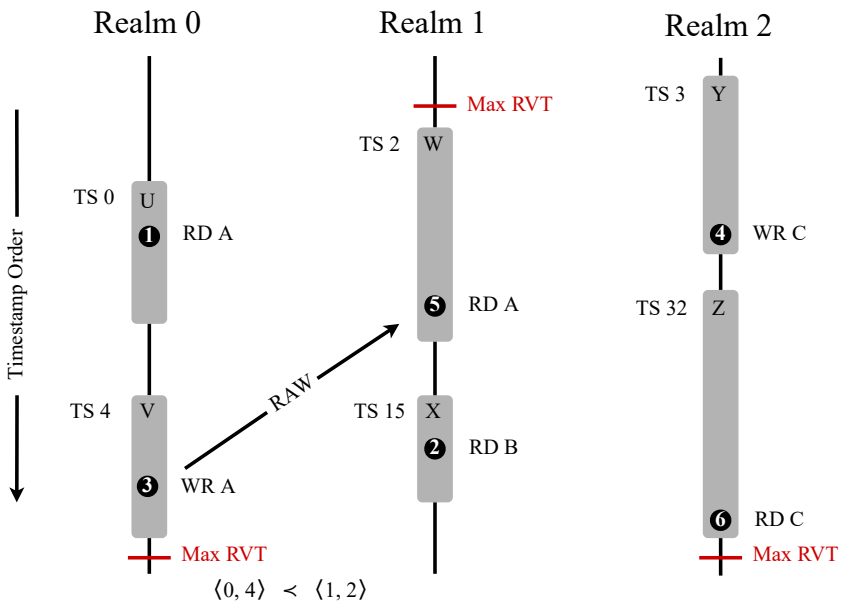
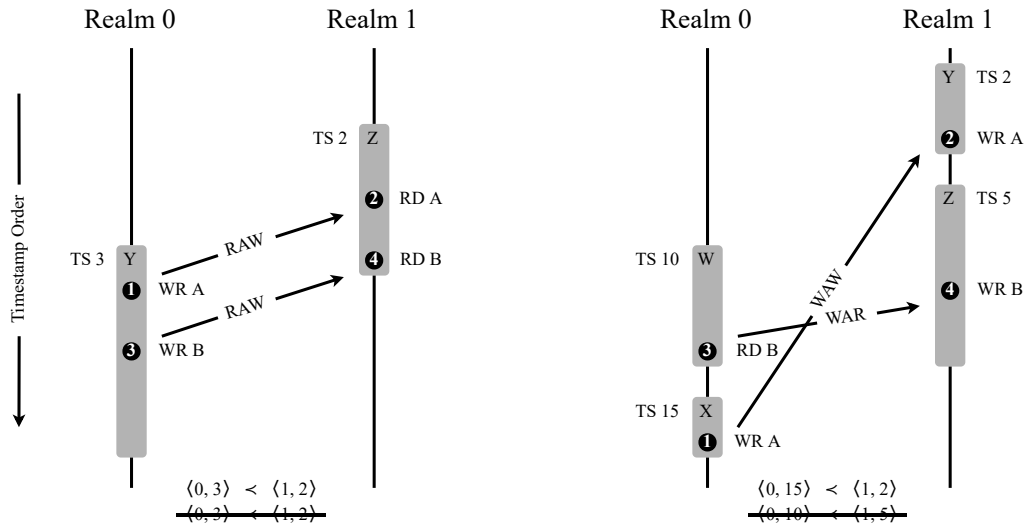


Figure 4-8: Scalable ordered commits in Nexion.

the same all-to-one reduction and one-to-all broadcast over trees of RVT arbiters as described in Section 4.1.3. Instead of broadcasting the minimum timestamp of all unfinished tasks in the realm, the root RVT arbiter must also examine the imposed happens-before orders on timestamps of different realms. In particular, the RVT cannot pass a timestamp that might depend on speculative data from another realm as the task that produced the speculative data may still abort.

For example, consider Figure 4-8. Since task W read speculative data from unfinished task V in a different realm, the maximum RVT for realm 1 cannot pass timestamp 2. Only once the RVT in realm 0 passes timestamp 4, causing task V to commit, can task W commit. Since realm 0 is only the source of a speculative data dependency and does not itself depend on other realms, the RVT is free to advance without limit. Furthermore, realm 2 is not involved in any speculative data dependencies so it also has no limitation on its RVT and can commit tasks independently from realm 0 and 1. This maintains that realms which do not share speculative data can execute completely concurrently, without synchronization.

The root RVT arbiters in Nexion determine the maximum RVT by scanning the



(a) Duplicate dependencies.

(b) Redundant dependencies.

Figure 4-9: Reducing tracked dependencies.

queues of incoming happens-before dependencies, looking for the one with the minimum destination timestamp. In Figure 4-8, this is $\langle 0, 4\rangle \prec \langle 1, 2\rangle$. The destination of this dependency, $\langle 1, 2\rangle$, represents the earliest timestamp in realm 1 that depends on speculative data from another realm. Note that the maximum RVT calculation only depends on incoming happens-before dependencies for the realm.

Upon an RVT advancement, the root arbiter must clear outgoing happens-before dependencies for the realm. This process notifies other realms that the sources of speculative data have committed and their maximum RVT limit can increase, allowing for RVT advancement. In Figure 4-8, realm 0's RVT will eventually advance past timestamp 4. Once this happens, realm 0 will remove the outgoing happens-before dependency $\langle 0, 4\rangle \prec \langle 1, 2\rangle$ and realm 1 is free to advance its RVT past timestamp 2.

4.4 Reducing Tracked Dependencies

As the first step towards a realistic hardware implementation, Nexion exploits the observation that many speculative data dependencies yield redundant happens-before order constraints. Given this observation, the number of explicitly tracked dependen-

cies can be reduced, freeing space in the queues.

The first kind of redundant dependency is simple duplicates. For example, consider Figure 4-9a. Tasks Y and Z first form a speculative data dependence on address A, establishing the happens-before order $\langle 0, 3\rangle \prec \langle 1, 2\rangle$. Both tasks then form the same speculative data dependence on address B. Even though this is a data dependence on a different address, it establishes the same happens-before order on timestamps. As a result, it is unnecessary to track both.

The second kind of redundant dependency arises when one happens-before relation logically implies the other. In this case, one dependency is essentially stronger than the other and it is sufficient to only track one to retain the same information. A happens-before dependency $\langle r_{src}, t_{src,strong}\rangle \prec \langle r_{dst}, t_{dst,strong}\rangle$ is at least as strong as $\langle r_{src}, t_{src,weak}\rangle \prec \langle r_{dst}, t_{dst,weak}\rangle$ if $t_{src,strong} \geq t_{src,weak}$ and $t_{dst,strong} \leq t_{dst,weak}$. In other words, a stronger dependency has a later source timestamp and an earlier destination timestamp than a weaker dependency.

For example, consider Figure 4-9b. Tasks X and Y form a speculative data dependence on address A and tasks W and Z form a dependence on B. The first dependence establishes the happens-before order $\langle 0, 15\rangle \prec \langle 1, 2\rangle$. However, the second happens-before order, $\langle 0, 10\rangle \prec \langle 1, 2\rangle$, is implied by the first. Since timestamps within each realm are always comparable, there exist implicit happens-before orders. In particular, $\langle 0, 15\rangle \prec \langle 1, 2\rangle$ does not only state that task X happens-before task Y, it also states that any task before task X happens-before any task after task Y. Therefore, the entire transitive happens-before order established from task X and Y's speculative data dependence is $\langle 0, 10\rangle \prec \langle 0, 15\rangle \prec \langle 1, 2\rangle \prec \langle 1, 5\rangle$. This implicit transitive dependence encompasses the explicit dependence $\langle 0, 10\rangle \prec \langle 1, 5\rangle$ and is thus redundant.

Not all happens-before dependencies are redundant. Consider the example in Figure 4-10. Here, both $\langle 0, 15\rangle \prec \langle 1, 5\rangle$ and $\langle 0, 10\rangle \prec \langle 1, 2\rangle$ are unique as neither implies the other. Seen differently, it is possible to introduce an additional happens-before relation between these two, such as $\langle 1, 4\rangle \prec \langle 0, 11\rangle$, without conflict. Even though these dependencies are not redundant, it is possible to replace both of these happens-before orders with a single, stronger order. Given two dependencies $\langle r_{src}, t_{src,a}\rangle \prec \langle r_{dst}, t_{dst,a}\rangle$

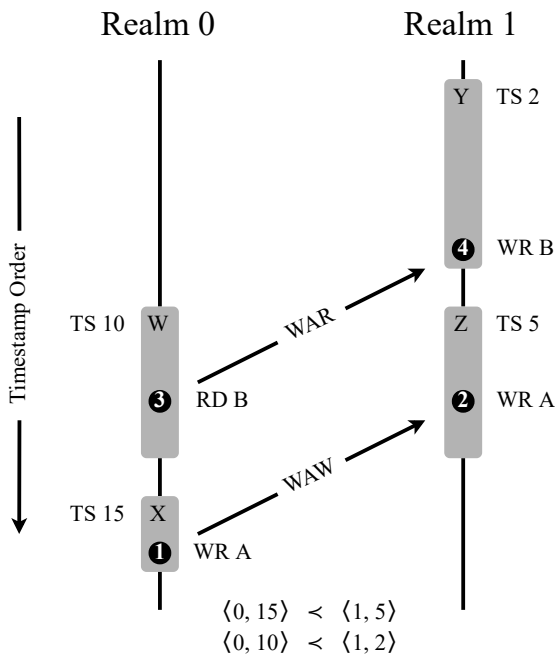


Figure 4-10: Necessary dependencies.

and $\langle r_{src}, t_{src,b} \rangle \prec \langle r_{dst}, t_{dst,b} \rangle$, they can be replaced with a single, stronger dependency $\langle r_{src}, \max(t_{src,a}, t_{src,b}) \rangle \prec \langle r_{dst}, \min(t_{dst,a}, t_{dst,b}) \rangle$. In this example, the replacement dependency would be $\langle 0, 15 \rangle \prec \langle 1, 2 \rangle$.

Unfortunately, this introduces false dependencies between timestamps. For example, the proposed additional happens-before relation above, $\langle 1, 4 \rangle \prec \langle 0, 11 \rangle$, can no longer be introduced without conflict. This loss of information triggers spurious aborts and can yield certain undesirable pathologies such as self-aborts. As a result, the current implementation of Nexion does not replace nonredundant dependencies and, for simplicity, only drops redundant dependencies if they share the same destination timestamp.

4.5 Handling Limited Queue Sizes

Happens-before dependencies in Nexion are stored in the root RVT arbiter in each realm. Each realm has a queue for outgoing dependencies to every other realm. Given the large number of dependency queues, their capacities must be kept small to

scale. By only replacing a subset of redundant happens-before dependencies and not replacing nonredundant dependencies, the dependency queues may reach capacity. As a result, a policy must be in place to handle that occurrence without sacrificing correctness. In this implementation of Nexion, memory accesses that induce happens-before dependencies when the respective queue is full simply stall until there is room. This policy is sufficient as accesses to data speculatively shared across realms are rare, and thus stalls occur infrequently. However, stalls present potential forward progress issues that must be handled with care.

In Swarm, forward progress is guaranteed by a total ordering of all tasks. There is always a task with the smallest timestamp. This task, called the GVT task, cannot be aborted and thus is considered irrevocable. Because this task cannot be aborted, it will eventually commit, leaving the task with the next smallest timestamp to run irrevocably, and so on. To apply the same forward progress argument for Nexion, it must be shown that there is at least one task with effectively the smallest timestamp in the system that cannot be aborted nor stalled. Once it commits, it must leave at least one task with the smallest timestamp to run irrevocably, and so on.

Since Nexion does not maintain a total order among all tasks, it cannot guarantee forward progress by prioritizing the earliest task as done before. Instead, Nexion only has a total order within each realm, and within each realm, there is an earliest task that we call the RVT task. The problem is that multiple realms can acquire order relations dynamically so there isn't a clear GVT task among these RVT tasks. A simple solution to ensure forward progress is to ensure that RVT tasks in all realms are prioritized and cannot be stalled by non-RVT tasks. Spatial partitioning makes this simple.

4.6 Filtering Realm Conflict Checks

Nexion recognizes that checking all tasks for conflicting accesses on every memory access is impractical. Like in Swarm, Nexion provides several mechanisms that leverage the cache hierarchy to filter conflict checks and dramatically reduce overheads. In

particular, realms in Nexion are established as partitions of tiles. As a result, a conflict across realms is only possible upon an cache coherence invalidation request across tiles, implying a miss in the L2 cache. Thus, all mechanisms presented by Swarm in Section 4.1.1 regarding L1 and L2 conflict check filtering remain fully utilized in Nexion. Notably, hits in the L1 cache and hits in the L2 cache with a timestamp higher than the canary do not require conflict detection across realms. This ensures that Nexion has no overhead for the common case when tasks access local data.

In the case where an access misses in the L2 or has a timestamp lower than the canary, conflict detection across realms is necessary. As described in Section 4.3.1, this involves communication with the root RVT arbiter, which maintains the happens-before dependencies for the realm. The RVT arbiter will scan the dependency queues, communicating with other root RVT arbiters, to determine if the access introduces a conflict by checking for cycles in the happens-before graph. This process is described in detail in Section 4.7.

To reduce the occurrence of cycle detection across root RVT arbiters, realm conflict checks can be further filtered. Suppose a new speculative data dependence across realms forms an associated happens-before relation $\langle r_{src}, t_{src}\rangle \prec \langle r_{dst}, t_{dst}\rangle$. A conflict is only possible if a cycle forms in the happens-before dependency graph. Given the graph is guaranteed to be a DAG before this edge is added, if there is a cycle, it must contain the newly-added edge. Therefore, a cycle is only possible if there exists some happens-before relation $\langle r_{dst}, t\rangle \prec \langle \cdot, \cdot \rangle$ such that $t_{dst} \leq t$. To avoid cycle detection, each tile maintains an outgoing upper bound timestamp t_{ub}. This timestamp is the maximum source timestamp for all outgoing happens-before within the realm. Now, for the new happens-before dependency, if $t_{dst} > t_{ub}$ (which implies $\forall t.t_{dst} > t$), then a conflict is impossible and thus cycle detection can be safely skipped.

To maintain this upper bound, if a tile introduces a new outgoing happens-before dependency with a source timestamp higher, the new upper bound must be distributed to all tiles in the realm. The one-to-all broadcast can efficiently reuse the existing RVT arbiter tree. When this upper bound is necessary, the overhead saved by avoiding a cycle detection outweighs the cost of maintenance.

Algorithm 1 Cycle detection.

```
1: procedure DetectCycle(D, ⟨r_src, t_src⟩ ≺ ⟨r_dst, t_dst⟩)
2:     M ← {r ↦ ∞ | ∀r ∈ R}
3:     return DetectCycleRec(D, ⟨r_src, t_src⟩, ⟨r_dst, t_dst⟩, M)
4: end procedure
5:
6: procedure DetectCycleRec(D, ⟨r_target, t_target⟩, ⟨r_current, t_current⟩, M)
7:     if t_current < M[r_current] then
8:         M[r_current] ← t_current
9:         for ∀r_next ∈ R do
10:            t_next ← MinDstAfter(D, ⟨r_current, t_current⟩, r_next)
11:            if r_next = r_target ∧ t_next ≤ t_target then
12:                return True
13:            else if DetectCycleRec(D, ⟨r_target, t_target⟩, ⟨r_next, t_next⟩, M) then
14:                return True
15:            end if
16:        end for
17:    end if
18:    return False
19: end procedure
```

4.7 Distributed Realm Conflict Detection

A modified depth-first search (DFS) cycle detection algorithm is used by Nexion to find conflicts in the happens-before graph. Prior work detects cycles in data dependencies without explicitly performing a graph traversal, though these methods introduce unnecessary serialization [17,21,25]. For example, in DATM [25], whenever a transaction forwards data to another transaction, an entry containing the source and destination of the dependency is appended to an order vector. To check for cycles, the order vector can be scanned to see if the current transaction ID is present earlier in the order vector. However, this technique imposes a serial order for all transactions with any data dependencies. This causes false cycles and unnecessarily serializes independent transactions with disjoint data dependencies. Furthermore, this only provides a conflict-serializable order of transactions, while Nexion must provide a serializable order of tasks that also respects program-specified timestamp order. Thus it is insufficient to simply check if a transaction ID exists in the order vector. An alternative implementation of DATM relies instead on ascribing times-

tamps to transactions, which is similar to LogTM [21]. DATM prevents cycles by only allowing earlier timestamped transactions to forward data to later transactions. In Nexion, this requires predetermining a total order on timestamps of all realms, which is undesirable as it limits concurrency. The cycle detection process used by Nexion is outlined in Algorithm 1.

Nexion establishes a partial order of all timestamps in the system. As described in subsection 4.3.1, this partial order can be viewed as a graph. To find cycles in a graph, a common strategy is to employ DFS. The standard DFS cycle detection algorithm recursively traverses a graph, maintaining a stack of nodes along the current path. If a node is reached that is already present in this stack of nodes, a cycle has been found. To avoid traversing the same parts of the graph again, a set of all visited nodes is also maintained. This algorithm can be applied without modification to the partial order graph in Nexion. However, there are several properties of Nexion's happens-before graph that can be exploited to reduce overhead.

Conflict detection is initiated upon the formation of a new speculative data dependency across realms and the associated happens-before relation $\langle r_{src}, t_{src}\rangle \prec \langle r_{dst}, t_{dst}\rangle$. Nexion guarantees that the set of all happens-before dependencies, D, up to this point form a DAG and must be cycle-free. Therefore, if this new dependency introduces a cycle, that cycle must go through the new dependency. Similar to Section 4.6, this means a cycle is only possible if there exists some happens-before relation $\langle \cdot, \cdot\rangle \prec \langle r_{src}, t\rangle$ such that $t \leq t_{src}$. Therefore, maintaining a stack of nodes along the current path is unnecessary as a cycle is only possible if a path reaches this point, denoted as $\langle r_{target}, t_{target}\rangle$ in Algorithm 1.

Furthermore, timestamps within the same realm follow a total order. This implies that a node in a realm has an edge to every node in the same realm with a higher timestamp. Therefore, when visiting a timestamp node in a realm, rather than traversing edges, it is sufficient to examine all outgoing edges with source timestamp greater than or equal to the current timestamp and find the minimum-destination timestamp edge to each other realm. This is accomplished using the MinDstAfter procedure in Algorithm 1. Then, all timestamps in the current realm greater than the

current timestamp can be marked visited. In Algorithm 1, this is maintained using the maximum-visited timestamp map M.

To implement this in hardware, each root RVT first initializes their maximum-visited timestamp map M to infinity. Then realm r_{dst} initiates the detection process by examining all outgoing dependencies to find the minimum-destination timestamp edge to every other realm with source timestamp greater than or equal to t_{dst}. For each call to DetectCycleRec, a message is sent to the respective realm with the calculated minimum-destination timestamp. Upon receiving this message, the next realm continues recursively until a path reaches $\langle r_{target}, t_{target} \rangle$ or a previously visited timestamp. For each return from a recursive call, a message is sent back to the requesting realm indicating if a cycle has been detected. To reduce complexity, and because cycle detection is rare, Nexion sends all messages across realms in series, waiting for a response before continuing. The primary benefit of this cycle detection method is it only involves communication between realms that are currently sharing speculative data.

4.8 Analysis of Hardware Costs

To implement Nexion, only minor changes are made to the Swarm baseline hardware. Namely, messages in the cache coherence protocol must include additional metadata and root RVT arbiters must maintain additional state.

To enable hierarchical conflict detection, Swarm modifies directory-based cache coherence protocol messages to include timestamp information. Nexion needs to add realm identifiers to timestamps the make them comparable. With this requires $\log(n)$ bits with n realms. For example, with a system of 16 realms, only fours bits need to be added to coherence messages.

Each root RVT arbiter maintains a set of queues for happens-before dependencies among realms. There is exactly one queue for every pairwise permutation of realms. In other words, if there are n realms, each root RVT arbiter will have $n - 1$ queues, one for outgoing happens-before dependencies to every other realm, excluding itself.

This makes a total of $n(n-1)$ queues in the system.

Each queue entry contains a single happens-before dependency, consisting of two 64-bit timestamps. If there are m entries per queue, each queue consists of $128m$ bits or $16m$ bytes. In total, that leaves $n(n-1) \cdot 16m$ bytes of additional state in a system with n realms.

The queues must support sorted-list operations such as finding the minimum-destination dependency or finding all dependencies with a source greater than a specified timestamp. Given these queues are quite small, specialized hardware such as TCAMs is unnecessary. Instead, the queues are implemented using registers and parallel comparators.

For a system with 256 cores, 16 realms, and four dependencies per queue, like the one evaluated in Section 5, Nexion requires less than 16 KB of state. This is less area than a single L1 cache. In summary, the hardware overheads to Nexion are moderate, and, in return, confer significant speedups.

Chapter 5

Evaluation

5.1 Experimental Methodology

5.1.1 Modeled System

We use a cycle-accurate, event-driven simulator based on Pin to model Nexion systems of up to 256 cores, as shown in Figure 4-5, with parameters in Table 5.1. Swarm parameters (task and commit queue sizes, etc.) match those from prior work [18]. The simulator uses detailed core, cache, network, and main memory models (derived from zsim [26]), and faithfully simulates all speculation overheads (e.g., running misspeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). The 256-core configuration is similar to the Kalray MPPA [8]. Smaller systems are also simulated with square meshes ($K \times K$ tiles for $K \leq 8$). The L2/L3 sizes and queue capacities per core are kept constant across system sizes. This captures performance per unit area. As a result, larger systems have higher queue and cache capacities.

5.1.2 Benchmarks

The primary benchmark used to evaluate Nexion is Silo, a recent, highly-scalable in-memory OLTP database [30]. This application was selected as transactional databases have both concurrency across transactions and ordered parallelism within transac-

Cores	256 cores in 64 tiles (4 cores/tile) partitioned into 16 realms (4 tiles/realm), 2 GHz, x86-64 ISA; Haswell-like 4-wide OoO superscalar
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 caches	64 MB, shared, static NUCA [19] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	8x8 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [31])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (16284 total), 16 commit queue entries/core (4096 total), 4 dependency queue entries/realm-pair (960 total)
Swarm instrs	5 cycles per `enqueue`/`dequeue`/`finish_task`
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [6], Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared, Realm checks add cycles for messages between arbiters
Commits	Tiles send updates to RVT arbiters every 200 cycles
Spills	Coalescers fire when a task queue is 85% full Coalescers spill up to 15 tasks each

Table 5.1: Configuration of the 256-core system.

tions; Nexion exploits both. This particular parallelism signature is far from specific to transactional database applications. Porting additional applications is left to future work.

Baseline Silo: Silo can be used to implement any generic, relational database and its associated transactions. Each table and index is implemented using a B+ tree with variable-length keys inspired by Masstree. The unique fields of a table are concatenated in big-endian order and used as keys in the B+ tree with the remaining fields as values. To provide concurrent access, the B+ tree uses a complex, fine-grain synchronization strategy.

Transactions in Silo are implemented using two phases, leveraging optimistic concurrency control (OCC) to provide high throughput. In the first phase, B+ tree operations are performed to read and compute updated values, which are locally buffered. Also at the time of access, B+ tree node and value version numbers are logged. In the second phase, locks of updated nodes and values are acquired and version numbers are compared. If version numbers remained the same, writes are applied, locks are released, and the transaction commits. Otherwise, the transaction aborts and is re-executed.

The transactional workload is evenly distributed to n worker threads, each mapped to a single core. When possible, to improve scalability, tables, and the transactions operating on those tables, are sharded across workers. Each worker serially processes a single transaction at a time.

Nexion Silo: To evaluate on Nexion, we port Silo to effectively exploit available concurrency across transactions and ordered parallelism within transactions. Tables and indexes are still implemented using B+ trees with variable-length keys. However, given Nexion's support for atomicity, the complex locking scheme is no longer necessary and thus is completely removed. Each B+ tree operation now occurs within a single atomic task. This provides stronger consistency guarantees than the baseline Silo implementation for B+ tree scans, which were previously not atomic.

Transactions in Nexion exploit ordered parallelism by separating B+ operations into individual tasks. If the order of B+ operations is important, this is reflected using timestamps. And if data must be passed across operations, this is done via continuation-passing style. The commit phase of a transaction can be implemented without locks in two different ways: using a single task or multiple tasks. The former implementation is called Nexion Single and the latter Nexion Multi. In both cases, version numbers are logged upon access and writes are locally buffered. In the single task case, version numbers are compared and all writes are applied in a single task relying on Nexion's atomicity guarantees across realms. However, this unnecessarily serializes the writes of a transaction, limiting scalability, and makes task aborts more costly (more work needs to be redone). To address this, writes can be split into individual tasks. However, since Nexion does not yet provide atomicity for groups of tasks across realms, because porting Fractal to Nexion is left for future work, this implementation requires transactions only access data sharded to their particular realm. This distinction does not reflect a limitation on the ideas presented by Nexion, rather on the current implementation. As a result, results using both methods will be presented.

Like the baseline, the workload is evenly distributed to n workers, each mapped to a single realm with tables and transactions sharded across workers. Each transaction uses a disjoint sequence of timestamps with respect to other transactions in the same realm to ensure atomicity when speculatively executing tasks of later transactions.

Microbenchmark: In addition to Silo, we perform sensitivity studies using a microbenchmark designed to stress the novel components of Nexion and demonstrate design trade-offs. The microbenchmark uses a shared counter. Each realm has a large amount of work to complete. This work consists of multiple stages, each with a random amount of work ranging from about 100 to 1000 cycles, that are completed sequentially. At the end of each stage, before moving onto the next, a counter shared by all realms is incremented. This increment occurs in a single task to guarantee atomicity.

Workload	Distribution
TPC-C	45% New Order, 43% Payment, 4% Delivery, 4% Order Status, 4% Stock Level
TATP	35% Get Subscriber Data, 35% Get Access Data, 14% Update Location, 10% Get New Destination, 2% Update Subscriber, 2% Insert Call Forwarding, 2% Delete Call Forwarding

Table 5.2: Transactional database workloads.

5.1.3 Database Workloads

Two different transactional database workloads are evaluated on Silo. Table 5.2 provides a summary of the workloads and their associated transaction distributions.

The Transaction Processing Performance Council Benchmark C (TPC-C) benchmark is modeled after a database that might be used by a wholesale supplier [1]. Tables in TPC-C maintain sets of warehouses, districts, items, customers, orders, etc. The scale factor varies the number of warehouses. Rows in each table belong to specific warehouses and thus can be sharded based on warehouse ID. Transactions are complex, consisting of queries to many different tables and have vastly different amounts of work. The Order Status and Stock Level transactions are read-only. The New Order and Payment transactions may involve two warehouses, with 1% and 15% probability, respectively (less than 7% of all transactions). Of all the workloads, this workload provides the most opportunity for exploiting intra-transaction parallelism.

The Telecommunication Application Transaction Processing (TATP) benchmark is modeled after typical telecommunication application [22]. Tables in TATP maintain sets of subscribers, facilities, call forwarding entries, etc. The scale factor varies the number of subscribers. Rows in each table belong to specific subscribers and thus can be sharded based on subscriber ID. Transactions can involve multiple queries, but are generally less complex than TPC-C transactions. To support queries based on subscriber number (as opposed to subscriber ID), a read-only index is created. The Get Subscriber Data, Get New Destination, and Get Access Data transactions are read-only. No transactions will involve more than one subscriber.

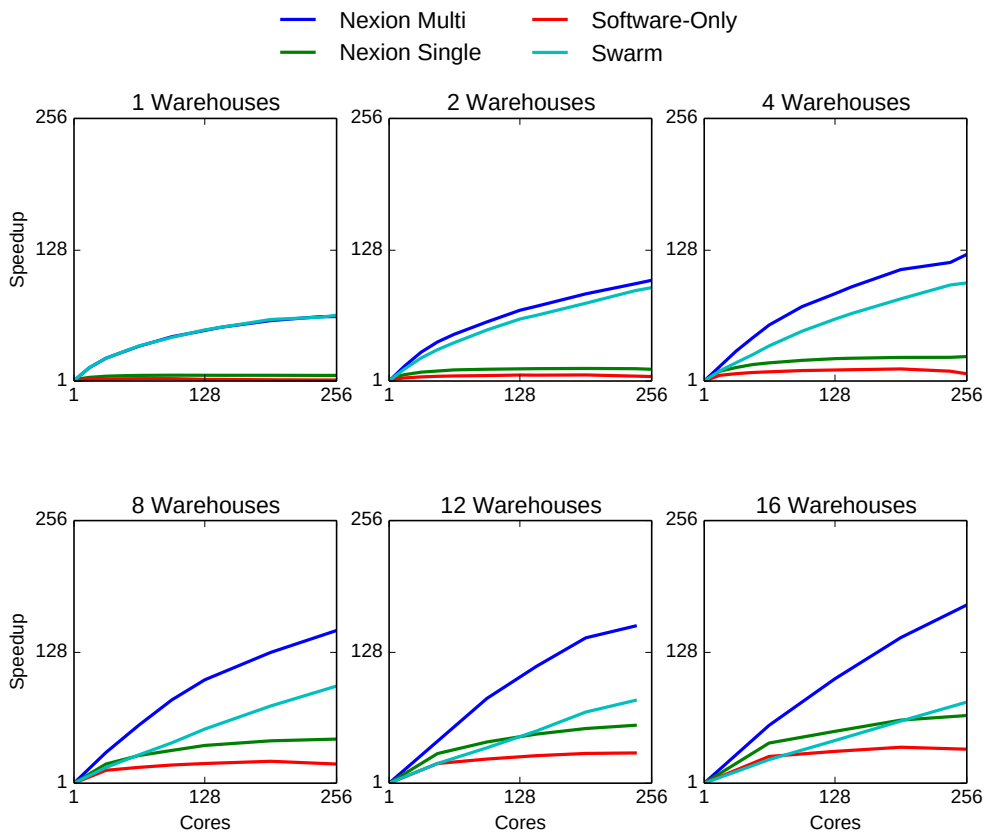


Figure 5-1: Speedup of Nexion on TPC-C across different warehouses.

5.2 Nexion Improves Scalability

Figure 5-1 compares the performance of Nexion, Swarm, and software-only implementations of Silo on the TPC-C workload with different number of warehouses. Each graph shows the speedup of the different implementations over the serial, software-only implementation of Silo running on a system of the same size, from 1 to 256 cores. Per-core L2/L3 capacities are kept constant as the system grows, capturing performance per unit area. For the Nexion benchmarks, realms are set equal to the number of warehouses.

Nexion Multi demonstrates the best scalability across all warehouses. In particular, Nexion Multi outperforms the software-only implementation 5.1-32×, and Swarm at most 2.4×. This section analyzes the performance of Nexion Multi against the software-only, Swarm, and Nexion Single implementations.

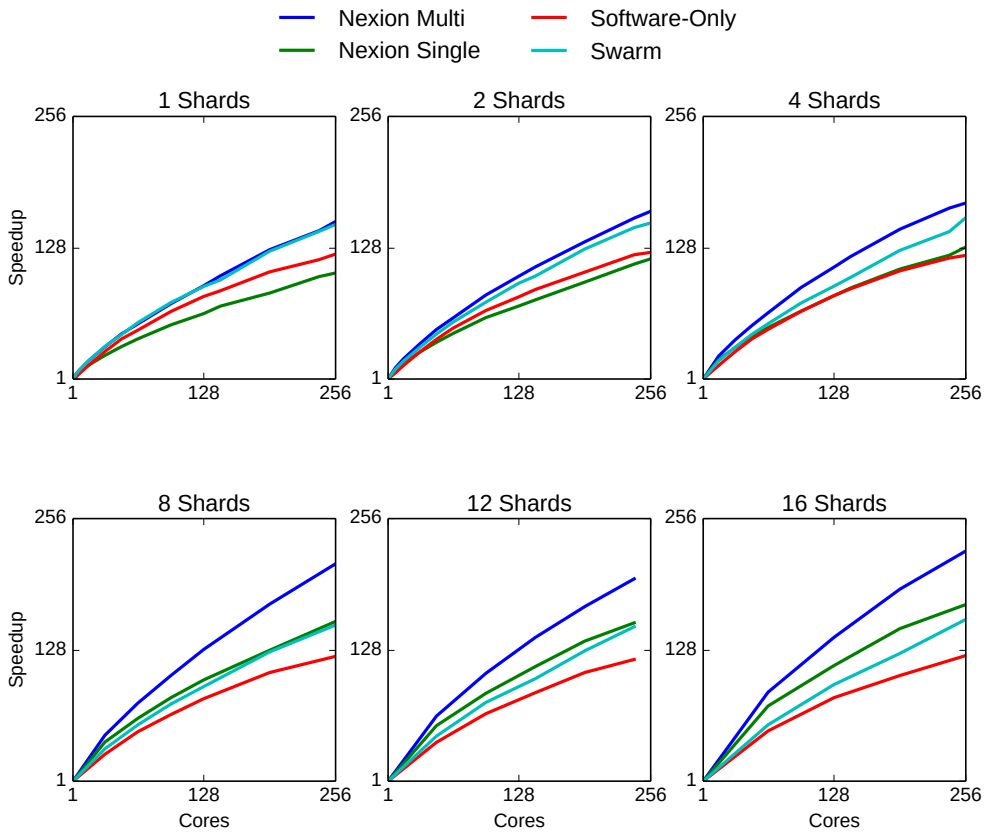


Figure 5-2: Speedup of Nexion on TATP across different shards.

In the case of 1 warehouse, Nexion Multi outperforms the software-only implementation by 32× but has identical performance to Swarm. This is due to the available ordered parallelism within transactions which is well-exploited by Swarm, but not as well-exploited by the software-only implementation. The software-only implementation is carefully tuned to achieve high transaction rates [30]. However, it is limited by the available concurrency among transactions. In TPC-C, this concurrency is determined by the number of warehouses. As a result, the software-only implementation scales linearly only when the number of cores is greater than the number of warehouses. With only 1 warehouse, transactions are aborted frequently, limiting scalability.

Swarm, on the other hand, takes advantage of the ordered parallelism found within transactions. Each task reads or writes at most a single tuple. This exposes parallelism within transactions, and reduces the penalty of conflicts, as only small, depen-

dent tasks are aborted instead of complete transactions.

Nexion Multi is able to match the Swarm implementation because it uses the same mechanism to exploit ordered parallelism within transactions. Just like Swarm, tasks read or write at most a single tuple. The only difference is that writes are locally buffered until commit. However, they are applied in separate tasks which exploits the same benefits as Swarm as conflicts only trigger small, dependent tasks to abort, not entire transactions.

Nexion Single loses most of the benefits of Swarm in the case of one warehouse. By applying all writes in a single task, there is a higher likelihood that tasks will conflict. Furthermore, upon a conflict, all writes will need to be reapplied. This emphasizes just how crucial it is to separate writes into individual tasks to enable scalability with few warehouses.

As the number of warehouses increases, so does the available concurrency. This enables the software-only and Nexion Single implementations to achieve slightly higher speedups. Swarm, on the other hand, loses ground as it does not exploit this additional concurrency. Instead, execution becomes dominated by larger tasks which effectively serializes execution through needless aborts, even under-performing the software-only implementation for small number of cores. Nexion Multi is able to effectively exploit both the available ordered parallelism with few warehouses and the additional concurrency with more warehouses.

Figure 5-2 shows the same information for the TATP workload. For Nexion benchmarks, realms are set equal to the number of shards. The TATP workload demonstrates similar characteristics as the TPC-C workload. However, there is much more concurrency among transactions, as they rarely accesses the same data, and less parallelism within transactions, as they consist of fewer queries. As a result, the software-only and Nexion Single benchmarks scale better than in TPC-C. However, the overall trend is the same as TPC-C: Nexion Multi is able to outperform all implementations by effectively exploiting available concurrency across transactions and ordered parallelism within transactions.

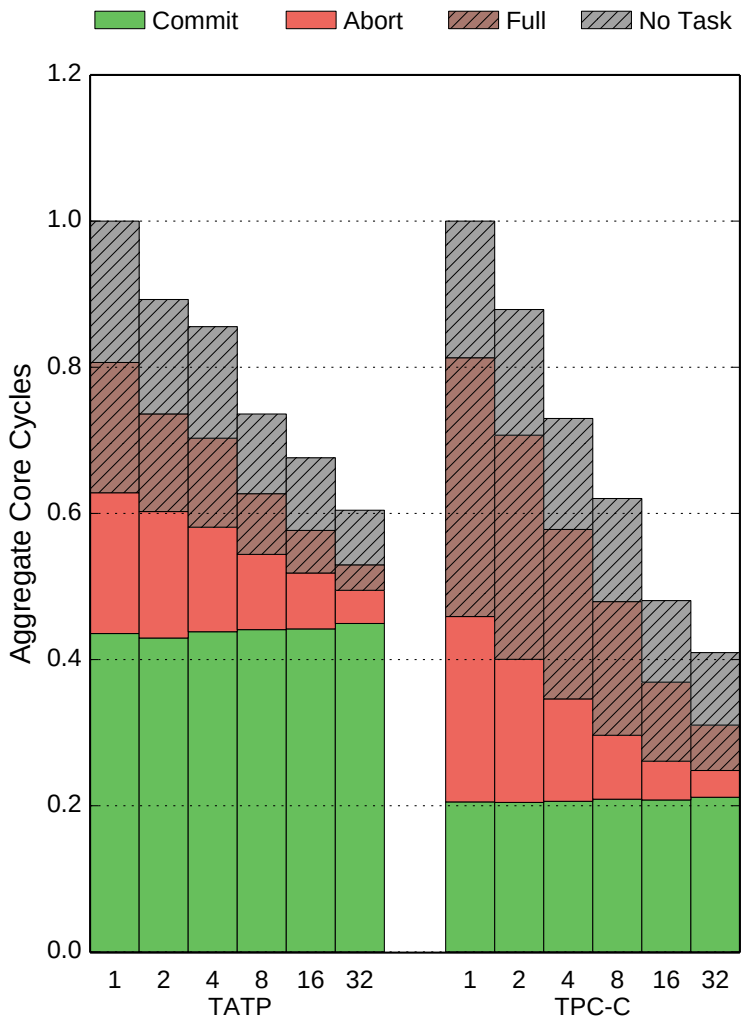


Figure 5-3: Breakdown of total core cycles for Nexion on TATP and TPC-C with different realms.

5.3 Nexion Reduces Pressure on Resources

Figure 5-3 shows the breakdown of aggregate cores cycles. Each set of bars shows results for a single workload as the system scales from 1 to 32 realms, holding the total number of cores constant at 256 cores. The height of each bar is the sum of cycles spent by all cores, normalized by the cycles of the single-realm system (lower is better). Lower bars indicate a reduction in the overall aggregate work performed by the system. Each bar shows the breakdown of cycles spent executing tasks that are ultimately committed, cycles spent executing tasks that are later aborted, and cycles

stalled either because the commit queue was full or no tasks were available. This analysis focuses on the TPC-C workload as it has more inter-transaction parallelism and thus higher speculation overheads than the TATP workload.

Regardless of the number of realms, the total amount of cycles spent executing tasks that are ultimately committed remains constant. However, with one realm, committed cycles make up only 21% of all executed cycles in the TPC-C workload. With 32 realms, committed cycles make up more than 52%. This improvement can be attributed to two sources. The first source of performance improvement stems from a reduction in the number of unnecessary aborts. With one realm, 25% of cycles are lost to aborted tasks. With 32 realms, this is reduced to less than 9%. These aborts stem from unnecessary order constraints. With one realm, database transactions have a total order. To maximize available parallelism, these transactions are executed speculatively out of order. However, if the transactions access the same data in an order different from their established order, one must abort. This happens even if the interleaving respects the atomicity requirements of database transactions, due to order constraints. With 32 realms, fewer transactions are ascribed an order and are thus permitted to interleave in more ways, reducing unnecessary aborts.

The second source of performance benefits comes from using fewer commit queue entries. With one realm, 35% of cycles are lost due to stalling on full commit queues in the TPC-C workload. With 32 realms, this is reduced to only 15%. Tasks in TPC-C have very skewed amounts of work. The most common transaction, New Order, is short and has a read and write set of at most 15. However, the Stock Level transaction is long and can have a read set greater than 200. With one realm, all transactions are ordered. Transactions can be executed out-of-order, but only while there is space in the commit queue. The large transaction bottlenecks the system by preventing later tasks from committing, filling the commit queue. With 32 realms, the order constraints are relaxed allowing transactions in other realms to proceed, even when a larger task might be consuming resources.

These two trends are further validated with the TATP workload, though it is less affected by these speculation overheads. In summary, Nexion reduces aborts and

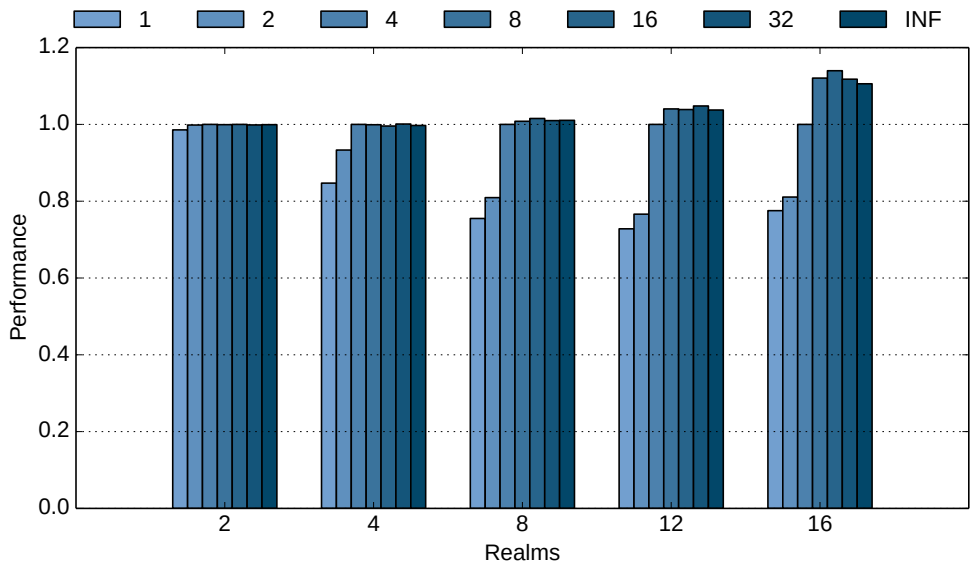


Figure 5-4: Sensitivity to dependency queue sizes with cores/realm constant.

frees up commit queue slots by removing unnecessary order constraints. This enables better scalability and more efficient use of resources.

5.4 Sensitivity Studies

Both the TPC-C and TATP workloads rarely access speculative shared data. The average dependency queue occupancy over all executed cycles exceeds 5%. As a result, this section explores Nexion's sensitivity to several design parameters using a shared counter microbenchmark to artificially introduce higher contention. In all cases, the number of cores per realm is held constant at four cores/realm.

5.4.1 Unlimited Dependency Tracking

Figure 5-4 shows the speedups of the shared counter microbenchmark as the size of the dependency queues varies from one to unbounded; the default is four entries. When the number of realms increases, the size of the dependency queues becomes more important. Using four entries is sufficient to stay within 10% of the performance with unbounded queues.

To get a sense of where the performance difference is coming from, Figure 5-5

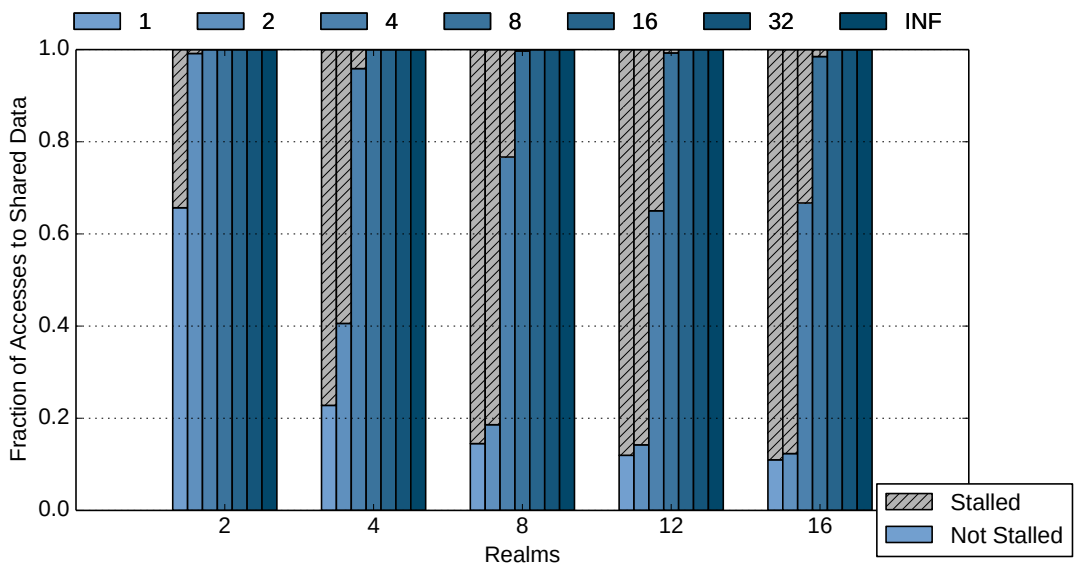


Figure 5-5: Effect of dependency queue sizes on stalls.

shows the fraction of accesses to shared data that result in a stall due to a full dependency queue. With more than two realms, having less than four entries in a dependency queue results in substantial number of stalls, reaching 89%. By increasing dependency queues to just four entries, the number of stalls reduces to less than 35%.

Furthermore, in attempt to estimate the optimal size of the dependency queues, consider a system with unbounded queues. Figure 5-6 shows the average number of dependency entries occupied over the duration of execution. With eight realms and fewer, the average number of occupied entries is less than four. With more than eight realms, the average number of occupied entries is slightly higher, but still less than six. This suggests more than four entries can moderately improve performance in the case of 12 and 16 realms, but four entries strikes a good balance between performance and implementation cost.

5.4.2 Instantaneous Realm Conflict Detection

Nexion introduces performance overheads of two main varieties: stalls and conflict detections. To get a sense of which contributes more significantly, cycle detection is replaced with an idealized verion which resolves realm conflicts instantly, within the same cycle as the triggering access. Figure 5-7 shows the speedups of the shared

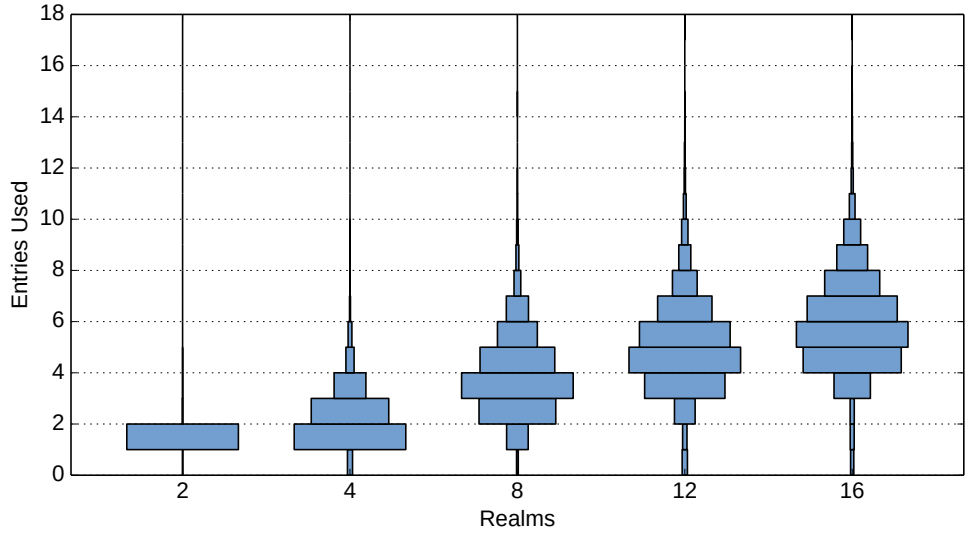


Figure 5-6: Average dependency queue occupancy.

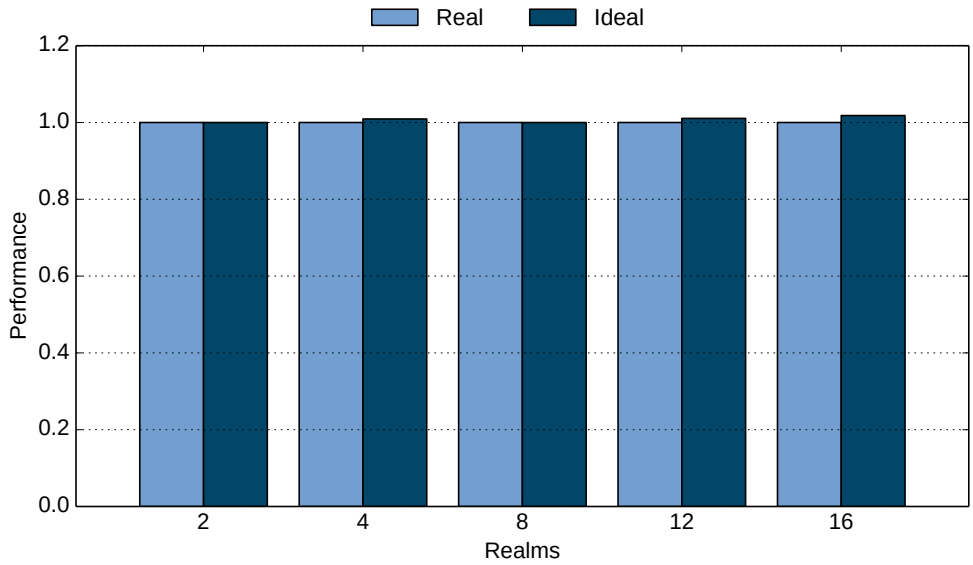


Figure 5-7: Realistic vs idealized cycle detection.

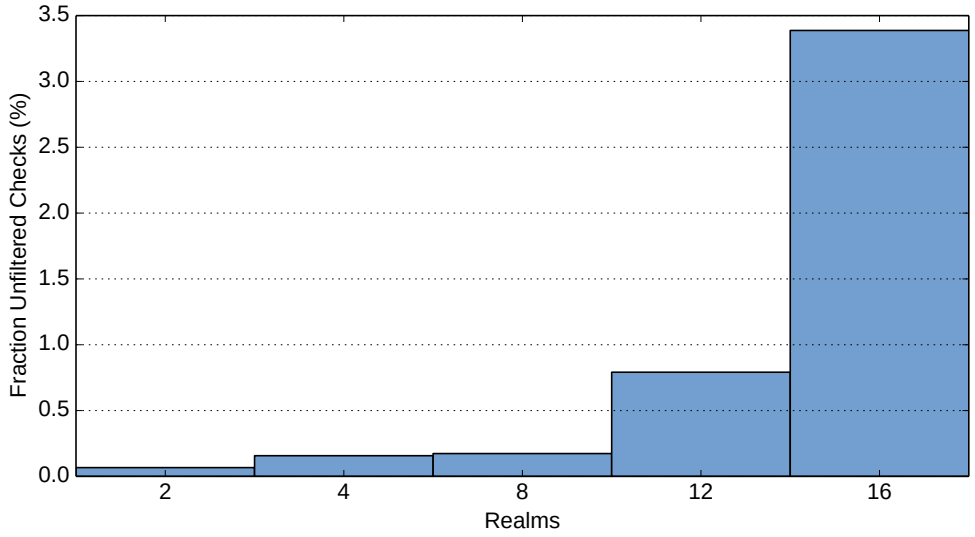


Figure 5-8: Fraction of cross-realm cycle checks that the upper-bound filter does not eliminate.

counter microbenchmark compared against an idealized conflict detection mechanism. The bars are normalized to the realistic conflict detection method. The flat bars suggest the conflict detection method is not a major source of overhead in Nexion.

5.4.3 Effects of Filtering Realm Conflict Checks

Nexion introduces an additional filtering technique to limit costly cross-realm cycle detection, as explained in Section 4.6. Figure 5-8 shows the fraction of realm conflict checks that necessitate cross-realm cycle detection in the presence of this filter. The filter dramatically reduces the number of cross-realm cycle detections such that, even with 16 realms, more than 96% realm conflict checks can prevented by maintaining an outgoing upper bound. This provides further evidence to suggest conflict detection is not a major source of overhead in Nexion.

Chapter 6

Conclusion

We have presented Nexion, a new execution model for harnessing concurrency alongside ordered parallelism. Nexion allows programmers to split applications into tasks and easily express available concurrency across tasks, while maintaining the original Swarm programming model crucial for scalably exploiting ordered parallelism. The presented implementation of Nexion avoids unnecessary serialization among tasks and only establishes order constraints between concurrent tasks when data is actually shared. This is accomplished with less area than a single L1 cache in a 256-core system and is fully decentralized, involving communication only between structures involved in data sharing. Nexion improves scalability up to 32× over software-only solutions and up to 2.4× over the original Swarm baseline.

6.1 Future Work

6.1.1 Diverse Benchmarks

Nexion has shown that providing architectural support for both concurrency and ordered parallelism in transactional databases can substantially improve scalability and efficiency across workloads. However, transactional databases are far from the only applications that have an abundance of concurrency and ordered parallelism. Additional benchmarks could be identified and ported to strengthen the argument for Nexion from a performance perspective.

6.1.2 Advanced Swarm Features

The Swarm baseline microarchitecture used in Nexion is from the original Swarm paper and is missing several key features that have been added by followup work in recent years. Most importantly, Nexion does not support features added by Fractal [29]. As mentioned in Section 5, this could immediately improve the performance and programmability of applications in Nexion. In addition, adding non-speculative tasks increases the practicality of the system and spatial hints improves performance within individual realms. Future work could investigate how to best extend the semantics of these features in the presence of realms.

6.1.3 Coarsening Tracked Dependencies

Tracking individual dependencies, even after applying reduction techniques discussed in Section 4.4, may become too costly for some workloads. Furthermore, Nexion currently requires serialization of updates to the queues in the root RVT arbiter. To reduce overheads and relax synchronization constraints in the implementation, Nexion can take inspiration from BulkSC [7] and investigate ways to coarsen dependencies. In particular, instead of establishing a happens-before relation on individual timestamps across realms, these relations could be established for ranges of timestamps. This could help amortize the cost of dependency tracking and allow for more localized conflict detection.

6.1.4 Multiprocessing

Instead of pursuing performance benefits for a single parallel application, as is the primary focus of our work, Nexion also has merits as a resource-sharing system. Without Nexion, Swarm can only execute a single application at a time. Supporting multiple processes is possible by utilizing time-division multiplexing, just as in a single-core system. However, Nexion provides additional flexibility by allowing multiple processes to share the system simultaneously. This makes Nexion more attractive as a general-purpose accelerator and opens up the possibility of utilizing novel hardware

features within a specialized operating system.

6.1.5 Dynamic Reconfiguration

Realms in Nexion are currently implemented as fixed partitions of cores. However, the execution model does not require this distinction. To improve flexibility, the presented implementation of Nexion could be extended to allow for dynamic repartitioning. Taking this idea further, it is also unnecessary for realms to be established even as disjoint partitions, they could instead overlap and share hardware resources without reconfiguration. This adds an additional design dimension for applications written using Nexion. In the case of multiprocessing, it also poses interesting opportunities for operating systems. Based on the current process utilization, the operating system can dynamically reallocate resources among processes. This also poses new challenges with identifying efficient scheduling of processes over time.

Bibliography

[1] TPC Benchmark C, 2010. `https://www.tpc.org/`.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, 2000.

[3] C. Scott Ananian, Krste Asanovic, Bradley Kuszmaul, Charles Leiserson, and Sean Lie. Unbounded transactional memory. volume 26, pages 316–327, January 2005.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Survey*, 13(2):185–221, June 1981.

[5] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[6] Larry Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC)*, 1977.

[7] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th annual International Symposium on Computer Architecture (ISCA-34)*, 2007.

[8] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA-38)*, 2011.

[10] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Computing Survey*, 49(2), June 2016.

[11] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[12] Maria J. Garzaran, Milos Prvulovic, and Jose M. Llaberia. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2:2005, 2005.

[13] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[14] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, 2004.

[15] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[16] Mark Hill and Michael Marty. Amdahl's law in the multicore era. *Computer*, 41:33 – 38, 08 2008.

[17] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. Wait-n-GoTM: Improving HTM performance by serializing cyclic dependencies. In *Proceedings of the 18th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, 2013.

[18] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO-48)*, 2015.

[19] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[20] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6:213–226, 1981.

[21] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: log-based transactional memory. In *Proceedings of the 12th IEEE international symposium on High Performance Computer Architecture (HPCA-12)*, 2006.

[22] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecommunication application transaction processing (TATP) benchmark description. `http://tatpbenchmark.sourceforge.net`, 2009 (accessed May 2022).

[23] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[24] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, 2005.

[25] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM international symposium on Microarchitecture (MICRO-41)*, 2008.

[26] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th annual International Symposium in Computer Architecture (ISCA-40)*, June 2013.

[27] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, 1995.

[28] J Greggory Steffan, Christopher B Colohan, Antonia Zhai, and Todd C Mowry. A scalable approach to thread-level speculation. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 1–12. ACM New York, NY, USA, 2000.

[29] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. Fractal: An execution model for fine-grain nested speculative parallelism. In *Proceedings of the 44th Annual International Symposium in Computer Architecture (ISCA-44)*, June 2017.

[30] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP-24)*, 2013.

[31] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, J.F. Brown, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE MICRO*, 27:15 – 31, 10 2007.

[32] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th IEEE international symposium on High Performance Computer Architecture (HPCA-13)*, 2007.

[33] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3):209–220, November 2014.