Evaluating the Feasibility of Transaction Scheduling via Hardware Accelerators

by

Thanadol Chomphoochan

S.B. Computer Science and Engineering
Massachusetts Institute of Technology, 2024

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 2025

Authored by:	Thanadol Chomphoochan Department of Electrical Engineering and Computer Science May 16, 2025
Certified by:	Adam Chlipala Professor of Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Evaluating the Feasibility of Transaction Scheduling via Hardware Accelerators

by

Thanadol Chomphoochan

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

As single-thread performance plateaus, modern systems increasingly rely on parallelism to scale throughput. Yet, efficiently managing concurrency—particularly in transactional systems—remains a major bottleneck. This thesis explores the feasibility of accelerating transaction scheduling via hardware, leveraging FPGAs to offload scheduling logic from the CPU. We revisit Puppetmaster, a hardware transaction scheduler, and present a redesigned architecture emphasizing deployability, modularity, and evaluation. We implement both an optimized software baseline and a Bluespec-based hardware design, evaluating their performance across synthetic YCSB-style workloads with varying contention levels. Our hardware prototype demonstrates competitive throughput, achieving over 90% of peak throughput even under high-contention workloads. These results validate the potential of transaction scheduling as a target for hardware acceleration and highlight promising directions for future hybrid hardware-software concurrency-control systems.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

I would like to thank Adam Chlipala for his guidance throughout this project. I'm especially grateful for his patience and accessibility—he always made time to answer my questions and discuss weekly progress updates. His flexibility, understanding, and trust were key to making this thesis possible. His class, Formal Reasoning About Programs, opened up an entirely new area of research for me and played a major role in shaping my academic direction.

Thank you to Thomas Bourgeat for teaching the class on Bluespec and for answering all my questions, whether basic or technical. He introduced me to this research topic, and his guidance helped ease many of my concerns as I worked through this thesis.

I also thank Áron Ricardo Perez-Lopez for his earlier work on Puppetmaster, which provided the basis for this project. Thanks to Elvis, Mingran, and others in CSAIL who helped set up machines and provided access to the hardware used in my experiments.

Thank you to Joe Steinmeyer, whose focus on education and low-level programming inspired my approach to learning and problem-solving. His encouragement and support, even when unrelated to this project, were fundamental to my growth in this area. I also appreciate his generosity in lending hardware and sharing advice whenever I needed it. Thanks also to Adam Hartz, my academic advisor, who has similarly been a steady source of guidance and support throughout my time at MIT.

I'm grateful to my closest friends—Arun Wongprommoon, Waree Sethapun, Joshua Nwakoby, Pranav Arunandhi, Nathan Shwatal—and many others I may have forgotten to mention here, for their encouragement throughout this process.

Finally, thank you to my family for their constant support. Their presence has always been a source of strength and stability.

Portions of this thesis were proofread and edited with the assistance of ChatGPT.

Any errors or omissions, if any, are entirely my own.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As a prelude to any discussions pertaining to parallel processing, it is customary to discuss Gordon Moore's prediction that the number of transistors in an integrated circuit would double about every two years. Many believe that Moore's Law, first formulated in 1965, no longer applies in modern times. With improvements in single-core performance becoming increasingly challenging, efforts have shifted toward leveraging parallel and distributed computations to enhance performance [14].

Parallel programming is, however, notoriously difficult. Since threads of computation can interleave unpredictably, assumptions about invariants may break in subtle yet catastrophic ways. Early primitives such as locks and atomic memory instructions allow threads of execution to be synchronized. However, the state space is often too large to reason through, making it challenging to design correct and performant parallel programs. Therefore, there have been many proposals for new abstractions to ease these pain points.

Transactions are a powerful abstraction in the database-system literature that allows the programmer to specify a sequence of commands to be executed atomically. The transaction engine guarantees that each transaction is fully executed or not executed at all. The engine also ensures other properties collectively called atomicity, consistency, isolation, and durability (ACID) to protect programmers from observing or having to manage unintuitive behaviors.

The concept of transactions, usually applied to the database setting, is so powerful that it has also been adapted as a synchronization primitive on a single multicore computer. This concept is called transactional memory (TM) [9]. Transactional memory allows the programmer to specify a sequence of instructions, including those that read from or write to shared memory, and have it be executed as an atomic unit or not executed at all. Unlike traditional lock-based approaches, TM shifts the burden of managing concurrency from the programmer to the runtime system, potentially improving both correctness and productivity.

While software implementations of TM offer flexibility, they often suffer from performance overheads due to instrumentation, bookkeeping, and validation costs. Hardware transactional memory (HTM) designs reduce these costs by providing direct architectural support for conflict detection and rollback. However, HTMs are typically constrained by hardware resources and may not support complex workloads or long transactions. Hybrid designs that combine both software and hardware techniques strike a balance between flexibility and performance. The push to use hardware to increase performance has broadened the design space of high-performance job scheduling and concurrency control in database systems. This

area of research remains ripe for exploration.

This thesis explores the feasibility of implementing transaction scheduling in hardware, leveraging FPGAs to accelerate scheduling decisions. We revisit Puppetmaster, a hardware transaction scheduler originally proposed in [19], and present a significantly redesigned architecture aimed at improving deployability, modularity, and evaluation. To assess its effectiveness, we implement an optimized software baseline and perform comparative benchmarking across various workloads. Our results demonstrate that with careful design, a hardware scheduler can achieve competitive performance, validating the potential of transaction scheduling as a viable target for hardware acceleration.

Chapter 2

Background

2.1 Transaction Scheduling

Transaction scheduling is the process of determining which transactions can be executed concurrently without violating correctness. It lies at the core of concurrency control in database systems and has also been explored in the context of transactional memory.

2.1.1 Concurrency Control

Concurrency-control mechanisms ensure that multiple transactions executing simultaneously do not lead to inconsistent states. Broadly, strategies fall into two categories: pessimistic and optimistic [16]. Pessimistic strategies prevent conflicts through mechanisms such as locking. Optimistic strategies assume conflicts are rare and detect them at commit time, rolling back the execution of transactions as needed [12]. Common implementations include timestamp ordering, deadlock detection, validation schemes, and serialization graphs.

The "Staring into the Abyss" paper [23] examines the scalability challenges of existing concurrency-control mechanisms as the number of cores increases. It finds that traditional techniques such as two-phase locking and optimistic concurrency control suffer from significant contention and overheads in multicore environments. These findings motivate the exploration of alternative architectures, including hardware-assisted transaction scheduling.

2.1.2 Transactional Memory (TM)

Software Transactional Memory (STM) [20] implements transactional semantics in software. STM systems track memory accesses in logs—either redo or undo logs—and validate transactions during commit. Systems may use eager or lazy versioning, and conflict detection may be implemented at various stages of transaction execution. Although STMs offer portability and ease of development, they introduce nontrivial overheads.

Hardware Transactional Memory (HTM) [10] provides architectural support for transactions, enabling faster conflict detection and rollback through cache-based mechanisms. However, HTMs are typically constrained by hardware limitations such as cache size and support for limited instruction types. Notable HTM implementations include Intel TSX [13] and AMD Advanced Synchronization Facility (ASF) [1].

Hybrid TM systems combine the strengths of STM and HTM. For example, Hardware-Assisted Transaction Scheduler (HaTS) [4] introduces software-computed "conflict indicators" which group potentially conflicting transactions into distinct scheduling queues for execution on an HTM system. While the grouping may not always be accurate, the underlying HTM system ensures correctness. By leveraging the HTM effectively, overall performance improves. HaTS exemplifies hybrid TM systems, in which the design space is not restricted to either software or hardware but instead requires their collaboration.

2.1.3 OLTP and TM Benchmarks

To evaluate transactional systems, standard benchmarks are used to simulate realistic workloads.

The TPC-C benchmark [22] models a wholesale supplier with multiple warehouses and transaction types. TPC-C transactions typically touch multiple objects with varying read and write patterns. This benchmark is commonly used to evaluate online transaction processing (OLTP) systems. The DBx1000 system [23] is an in-memory database that provides a reference implementation for TPC-C and other workloads. Key characteristics of TPC-C include skewed access patterns, large object spaces, and complex dependencies among transactions. These traits make it a useful testbed for evaluating the scalability and efficiency of transaction schedulers.

The STAMP benchmark suite [17] includes a set of applications tailored to evaluate TM systems. It includes workloads with various contention profiles and access patterns. The Stampede project [18] extends STAMP with more instrumentation and modifications to better simulate high-contention environments and hybrid memory models.

The Yahoo! Cloud Serving Benchmark (YCSB) [5] is another widely used benchmark, particularly for evaluating key-value and NoSQL stores. YCSB models a simple transactional workload on a single table with millions of rows. Each row contains fixed-size records, and the benchmark defines a single transaction type parameterized by access count (typically 16 records), the ratio of reads to writes, and the access pattern (e.g., uniform, Zipfian). Due to its simplicity and tunability, YCSB is well-suited for isolating the effects of scheduling and contention, and it forms the primary benchmark used in this thesis.

2.2 Field-Programmable Gate Arrays (FPGAs)

FPGAs are programmable logic devices that offer fine-grained control over hardware behavior. They are commonly used in domains where customized parallelism, deterministic latency, or offloading computation from CPUs is advantageous. Users write hardware designs in Hardware Description Languages (HDLs) such as Verilog, VHDL, or Bluespec.

2.2.1 Bluespec Hardware Description Language

Bluespec [2] is a high-level HDL based on guarded atomic actions. It provides a functional programming style and strong type system, which simplifies reasoning about concurrency in

hardware. Bluespec designs are compiled into Verilog, making them compatible with standard FPGA toolchains. Bluespec is our language of choice for implementing Puppetmaster.

2.2.2 Connectal Framework

Connectal is a codesign framework that integrates Bluespec-based hardware with C++ host software. It allows developers to declare interfaces in a makefile using a domain-specific syntax. For each declared interface, Connectal generates the necessary glue logic, including PCIe drivers and host-device bridges.

Connectal expects the project to implement a hardware top-level module that defines methods and rules corresponding to these interfaces. The framework injects host-facing interfaces into this module, allowing the host to invoke hardware functions via generated software bindings. Likewise, the hardware can invoke host functions using reverse interfaces.

Developers typically write host-side C++ code in the same project directory and register it in the makefile. Connectal then compiles both hardware and software components together. Running the compiled binary automatically uploads the bitstream to the FPGA and executes the host code.

While Connectal provides a streamlined development experience for simple projects, it imposes a rigid project structure. Its tight coupling between host and hardware interfaces makes it difficult to modularize or experiment with alternative I/O channels. These limitations are discussed further in later chapters.

2.2.3 Peripheral Component Interconnect Express (PCIe)

PCIe is a high-speed, point-to-point serial communication standard that connects peripheral devices to a host CPU. It supports multiple lanes for high-throughput data transfer and uses a layered protocol: the transaction layer (for packet generation), the data-link layer (for reliability), and the physical layer (for actual transmission).

In FPGA-based systems, communication over PCIe is typically mediated by a kernel driver or module that maps memory regions or exposes device-specific interfaces to user-space programs. Connectal provides such a driver, abstracting away low-level protocol details and exposing function calls for sending and receiving data.

Under the hood, communication often involves DMA (Direct Memory Access) engines, which allow data to be moved between host and device memory with minimal CPU intervention. The driver sets up descriptor queues that define memory transactions. These queues are used by the FPGA logic to initiate or complete transfers, often through memory-mapped I/O (MMIO) or doorbell registers.

While PCIe provides excellent throughput, its latency can become a bottleneck for fine-grained operations such as transaction-level scheduling. Understanding the trade-offs of using PCIe is critical when evaluating hardware accelerators in codesigned systems.

2.2.4 Heterogeneous CPU-FPGA Platforms

Modern platforms integrate CPUs and FPGAs into single packages to allow hardware-software co-execution. Examples include AMD's Zynq SoCs, which pair CPU cores with

programmable logic on a single chip. Cloud offerings such as Amazon EC2 F1 and F2 instances provide machines with high core counts connected to FPGAs via vendor-specific "shell" environments and high-bandwidth memory. These platforms support a wide range of deployment models for accelerating compute-intensive tasks, including transaction-scheduling engines like Puppetmaster.

Such systems enable experiments in offloading compute kernels to hardware while retaining the flexibility of general-purpose processors for control logic. As these platforms become increasingly accessible, they provide an ideal setting for exploring the feasibility of hardware-accelerated scheduling.

Chapter 3

Software and hardware system design for Puppetmaster

This chapter describes the full system architecture of Puppetmaster, covering both its software simulation and hardware implementation. We begin with an overview of Puppetmaster's role as a conflict-aware transaction scheduler and its interface with clients and executors.

We then detail the hardware communication pipeline and supporting development tools, which enable modular builds and benchmarking across different configurations. The software interface and simulator are introduced next, mirroring the hardware design to provide a fair and tunable performance baseline.

The latter sections focus on the hardware scheduler's architecture, including pipelining, conflict detection using Bloom filters, and techniques for lookahead and rescheduling. We conclude with an evaluation of Bloom filter design choices, such as chunking and filter separation, along with synthesis results and trade-offs.

Together, these components form the foundation for the performance evaluation in the next chapter.

3.1 Puppetmaster's responsibilities

Puppetmaster is a hardware-based transaction scheduler. Its input is a stream of transaction descriptors, and its output is a stream of scheduling decisions. Puppetmaster's role is to identify sets of transactions that can be executed concurrently without conflicts. It maximizes the number of concurrently executable transactions, while minimizing scheduling latency and maximizing throughput.

A transaction descriptor consists of a transaction identifier (transaction ID), a read set (the set of objects read by the transaction), and a write set (the set of objects read or written by the transaction). The descriptors are provided by the *host*, which is typically a software runtime on the host CPU.

For each transaction it receives, Puppetmaster emits a corresponding *start* message—though not necessarily in the order the transactions arrived. It only starts a transaction if it deems it safe to execute. These decisions are sent to the *executor*, a collection of worker cores that have the capacity to perform the actual work. These worker cores are referred

to as *puppets* in the original paper. The executor is responsible for notifying Puppetmaster upon the completion of a transaction. Figure 3.1 summarizes the intended use and high-level operation of Puppetmaster.

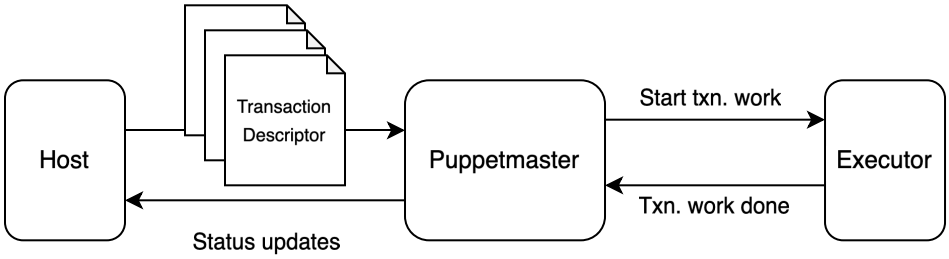

Figure 3.1: Abstract view of Puppetmaster

The notion of an object is opaque to Puppetmaster. Each object is identified by a 64-bit integer, to which Puppetmaster assigns no internal semantics. One may view Puppetmaster as a lock manager implemented in hardware. Puppetmaster guarantees only that, at any given time, all transactions it has started are either conflict-free or have already completed execution. Therefore, the host is ultimately responsible for partitioning the workload into transactions and correctly specifying the read and write sets, in order to maintain the desired application-level consistency guarantees.

3.2 Hardware communication interface

Puppetmaster is designed to be agnostic to both the source of transaction requests and the location of the execution units that carry them out. This flexibility allows the system to support a range of deployment configurations, including both software-driven and hardware-driven modes.

In typical use cases, the host CPU is responsible for issuing transactions to Puppetmaster. As such, the original implementation tightly integrated a Connectal-based communication interface between the host CPU and Puppetmaster. The Connectal interface allowed software running on the CPU to initiate and manage transactions over PCIe. However, due to the relatively high communication latency inherent to this setup, isolating the impact of the host interface on microbenchmarking became necessary.

To eliminate the influence of host-CPU communication latency on microbenchmarking, we introduced a hardware-based transaction generator that allows transaction requests to be driven directly from within the FPGA. While software is still used to configure the generator and trigger the start of the benchmark, the transactions themselves are emitted in hardware. This setup is generalized by abstracting the input driver into a reusable module interface. Because FPGA synthesis is time-consuming and input drivers generally occupy minimal area, we synthesize all potential drivers together and use a MUX—controlled by the CPU testbed—to select the active source. This modular design not only enables benchmarking under controlled conditions but also allows us to move away from the Connectal framework, opting for lighter-weight alternatives when needed.

On the execution side, Puppetmaster supports two modes. In the first mode, execution is performed on the host CPU, where transactions are routed back via PCIe. In the second mode, transactions are executed directly in hardware units on the FPGA. For benchmarking, these hardware units are currently "fake" executors that simply wait for a specified number of cycles, but the architecture is designed to support realistic hardware accelerators in future use cases.

Puppetmaster sends a transaction start message, also called a *work request*, which instructs the executor to begin processing the transaction. To support both execution modes without the need for multiple FPGA builds, we abstracted the executor interface in a manner similar to the input driver. Previously, the original implementation required toggling macro definitions and generating two separate software interfaces under Connectal to switch between execution modes—a cumbersome and error-prone process. Our abstraction avoids the need for multiple synthesis passes and provides a unified interface for transaction completion messages.

All communication channels between Puppetmaster and external components (such as the host CPU or hardware modules) are represented as external connections in Figure 3.2.

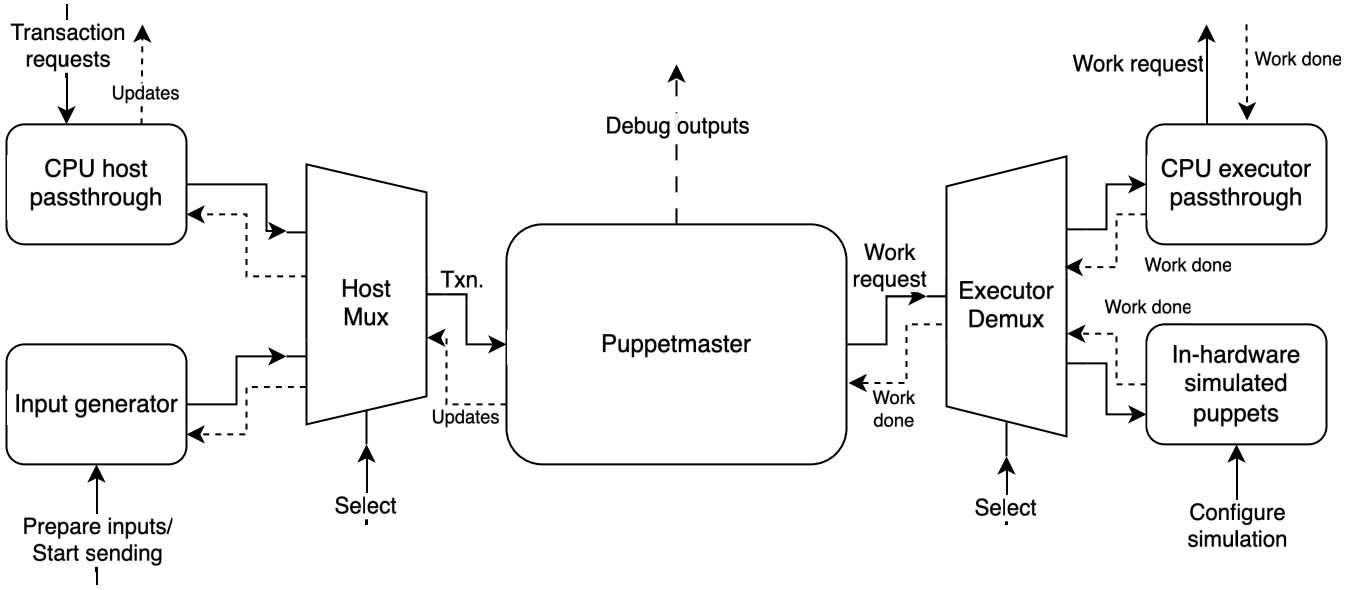

Figure 3.2: Block diagram for Puppetmaster hardware implementation

To understand the data flow within the system, consider the path of a single transaction. Initially, the host CPU software issues a transaction request. This request includes a transaction ID, a 64-bit user-defined auxiliary field (which may encode flags or parameters), and up to 16 associated objects. The transaction is then passed to Puppetmaster. Puppetmaster eventually decides to schedule the transaction. A work request containing the transaction ID and auxiliary data is forwarded to the executor. For CPU-based execution, this work request prompts the host runtime to carry out the transaction. Since Puppetmaster provides all necessary metadata, the runtime can perform the work independently. Once execution is complete, the CPU sends a *work-done* message, again identified by transaction ID.

In benchmarking scenarios, the sequence is slightly different. The host software first configures the input and execution MUXes and sets the parameters for the input generator.

After issuing a start command, the input generator begins streaming transactions into Puppetmaster. These transactions follow the same internal processing flow, except the executor in this case is a hardware unit, not the CPU. No additional communication with the CPU occurs during execution. The auxiliary field of the transaction is used to specify the duration (in cycles) for which the fake executor should simulate activity. For performance measurement, we capture timestamps from the debug output channel to compute latency and throughput metrics.

3.3 Development tools

As discussed in subsection 2.2.2, Connectal provides a set of interfaces that enable communication between host software and hardware accelerators. However, this framework imposes a strict project structure in which it must serve as the top-level build system. It orchestrates the compilation and execution of both hardware and software, aligning with its marketed goal of enabling "software-driven hardware development." Despite this premise, the practical use of Connectal reveals that much of the software and interface code is still generated based on the hardware source code, making the relationship between components tightly coupled and inflexible.

Tight coupling between Connectal's host interface and the rest of the system introduces several workflow challenges. First, any change to the hardware or software side often requires re-running the entire Connectal build flow, due to interdependencies among Connectal's internal logic, its generated files, and our own project-specific code. While selective rebuilding is technically possible, it demands detailed knowledge of Connectal's undocumented internals. As a result, even small changes can incur significant time costs.

Furthermore, Connectal's generated interfaces are not modular. If, for example, we wish to switch away from Connectal's PCIe-based communication to a different transport mechanism, every dependent software component must be restructured accordingly. This lack of abstraction creates a high barrier to experimentation and modular development.

To overcome these limitations, we developed an internal tool that extracts only the user-relevant components from a Connectal build. Specifically, it generates a clean header file and object file containing the necessary interfaces, which can then be compiled into larger software systems independently. The generated files allow software developers to treat the Connectal-generated code as a standalone library, calling into it only when needed and freeing them from having to engage with the full Connectal build environment.

Another significant limitation of Connectal is its poor support for configuration management. Modifying hardware parameters usually requires a full rebuild, with no built-in system to track or reuse previously synthesized configurations. Therefore, we created a build system that treats the project directory as the ground truth and maintains a set of versioned copies, each corresponding to a specific configuration. Users can request builds for any known configuration, and the script will verify whether the corresponding bitstream is up-to-date with the latest source changes. If not, it provides the option to trigger a rebuild. The tool also supports uploading bitstreams directly to the FPGA, streamlining the deployment process.

The combination of these tools enables clean separation of hardware and software development. As long as the hardware-software interface remains consistent, developers on both

sides can work independently. Hardware teams can experiment with architectural parameters without breaking software compatibility, while software developers can iterate on their logic without inadvertently triggering time-consuming hardware-recompilation steps.

3.4 Software interface

The software interface to Puppetmaster provides a simple API for initializing the system, submitting transactions, polling for scheduling decisions, and reporting transaction completion.

To begin, the user must call `pmhw_init` with the number of clients (transaction sources) and puppets (executors). While these numbers could be inferred dynamically, providing the counts upfront simplifies the software simulation backend.

The execution model is configurable via `pmhw_set_config` and `pmhw_get_config`, which allow toggling between real and simulated executors and adjusting how long fake executors simulate work.

Once initialized, clients submit transactions via `pmhw_schedule`. This function is blocking and may stall if the internal communication channel is full. Internally, it delegates to Connectal's driver interface, which handles PCIe buffering and protocol logic.

Puppets poll for available work using `pmhw_poll_scheduled`. When a transaction becomes ready, the function returns the associated transaction ID. After completing execution, the puppet must notify Puppetmaster via `pmhw_report_done`.

All API functions are designed to be efficient and easily integrable. Requiring explicit client and puppet IDs avoids implicit state tracking and enables high-performance SPSC queue-based simulation, as discussed in Section 3.5.

A summary of the software interface is shown in Table 3.1.

Function	**Description**
`pmhw_init`	Initializes the Puppetmaster runtime. Takes the number of clients and puppets. Must be called first.
`pmhw_set_config`	Sets runtime configuration (e.g., fake-executor mode, work duration). Takes a `pm_config_t` pointer.
`pmhw_get_config`	Reads the current configuration into a `pm_config_t` struct.
`pmhw_schedule`	Submits a transaction for scheduling. Blocks if the internal queue is full. Requires client ID and a pointer to a `txn_t`.
`pmhw_trigger_input_driver`	Manually triggers the input driver. Used in benchmarking setups only.
`pmhw_poll_scheduled`	Blocking call for puppets to retrieve scheduled transactions. Takes puppet ID. Returns transaction ID when ready.
`pmhw_report_done`	Notifies the system that a transaction has completed. Takes puppet ID and transaction ID.
`pmhw_shutdown`	Cleans up internal state and terminates the runtime.

Table 3.1: Software interface for interacting with Puppetmaster

3.5 Software scheduling model

To model Puppetmaster's behavior at a high level, we implemented a software simulator that captures its core design principles. It adopts the same greedy scheduling algorithm we will later implement in hardware (section 3.6), selecting the earliest conflict-free transactions for execution. While not cycle-accurate, the simulator reflects the same queuing structure and execution model as the hardware design, allowing us to establish a meaningful and fair performance baseline.

3.5.1 Thread architecture

Although the simulator supports multiple transaction sources, for evaluation purposes in chapter 4, we restrict the system to a single source of transactions. This source, referred to here as the *client thread*, plays the role of the transaction driver described in earlier sections: it is responsible for streaming transactions into the system using the software interface described in Table 3.1. The simulator also spawns one thread per executor, each corresponding to a puppet, and a single scheduler thread that simulates the logic of Puppetmaster itself.

Each of these threads is statically pinned to a dedicated CPU core to avoid scheduling interference and context-switching overheads. The scheduler thread is assigned to core 0, the main testbench thread occupies core 1, the client thread is pinned to core 2, and the puppet threads are assigned starting from core 3 onward. This setup ensures stable and reproducible timing results across runs. A summary of this core allocation is provided in Table 3.2.

Core ID	**Purpose**
0	Scheduler thread
1	Main thread
2	Client thread
3	Puppet #0
4	Puppet #1
⋮	

Table 3.2: Core assignments used in the software simulation

3.5.2 Scheduling loop

The software scheduling model mirrors the structure of a streaming hardware pipeline. Each client thread communicates with the scheduler via a single-producer single-consumer (SPSC) queue. Similarly, each puppet has two SPSC queues: one used by the scheduler to deliver scheduled transactions and another for reporting completions. These queues are implemented using lock-free ring buffers and are sized to ensure they never become the bottleneck under load. (See Appendix B.)

To keep track of which transactions are currently executing, the scheduler maintains a per-puppet active queue. This queue, implemented as a circular FIFO buffer, stores the full

transaction descriptors and is used for two purposes: to check for conflicts during scheduling and to verify transaction completions. FIFO ordering is critical for simplifying cleanup. When a puppet reports that a transaction has finished executing, the scheduler assumes that this transaction corresponds to the head of the active queue and dequeues it accordingly. This assumption is enforced by the puppet runtime, which processes transactions strictly in arrival order.

The core of the scheduler loop is a greedy algorithm. At each iteration, the scheduler inspects each client's input queue and performs a bounded lookahead to fetch a small number of transactions into a local buffer. For each transaction in the buffer, the scheduler checks whether it conflicts with any live transaction currently executing on any puppet. If no conflict is detected and at least one puppet has capacity to accept more work, the transaction is assigned to an available puppet, added to that puppet's active set, and enqueued into the corresponding schedule queue. Puppet selection is performed arbitrarily via a free-list mechanism. Fairness is not prioritized in this model; instead, the goal is to keep the puppets saturated with conflict-free work while minimizing scheduling latency.

Each puppet thread continuously polls for scheduled transactions. Upon receiving a transaction ID, it simulates execution by busy-waiting for a fixed number of CPU cycles. This duration is specified at runtime and can be configured to simulate realistic workloads with nonzero execution time, or set to zero to study the limits of scheduler throughput. After completing its simulated work, the puppet enqueues a completion message to the scheduler and waits for its next assignment.

3.5.3 Logging and analyzing

To capture system behavior in fine detail, the simulator includes a binary logging facility that records key lifecycle events for each transaction. These events include when a transaction is submitted, when it is scheduled, when a puppet begins and ends execution, and when the scheduler performs cleanup. To reduce measurement overhead, logging is performed with configurable sampling. Typically, only one out of every 2^k transactions is logged, where k is chosen at runtime. We have confirmed empirically that sampling does not affect throughput or latency at the rates we use for evaluation.

After a benchmark run, we use an external analyzer to validate correctness and extract performance metrics. This analyzer ensures that no two conflicting transactions were ever live at the same time, that each puppet executed transactions in FIFO order, and that all transactions submitted were eventually scheduled and completed. These guarantees allow us to use this simulator as a faithful model of the intended hardware behavior.

Ultimately, the goal of this software implementation is to serve as a fair baseline that is demonstrably free of major bottlenecks. By ensuring that the scheduler is able to sustain high throughput and low latency even under artificial zero-contention conditions, we establish that any performance advantage shown by the hardware implementation arises from its architectural strengths, not from avoidable software inefficiencies.

3.6 Hardware architecture

This section describes the hardware implementation of Puppetmaster and the scheduling algorithm it employs. While the high-level behavior mirrors the greedy approach used in our software simulator, the hardware design introduces additional pipeline staging and concurrency to minimize latency and maximize throughput. At its core, the architecture conservatively tracks active transactions using Bloom filters and leverages efficient buffering to implement lookahead and rescheduling. The architecture is illustrated in Figure 3.3.

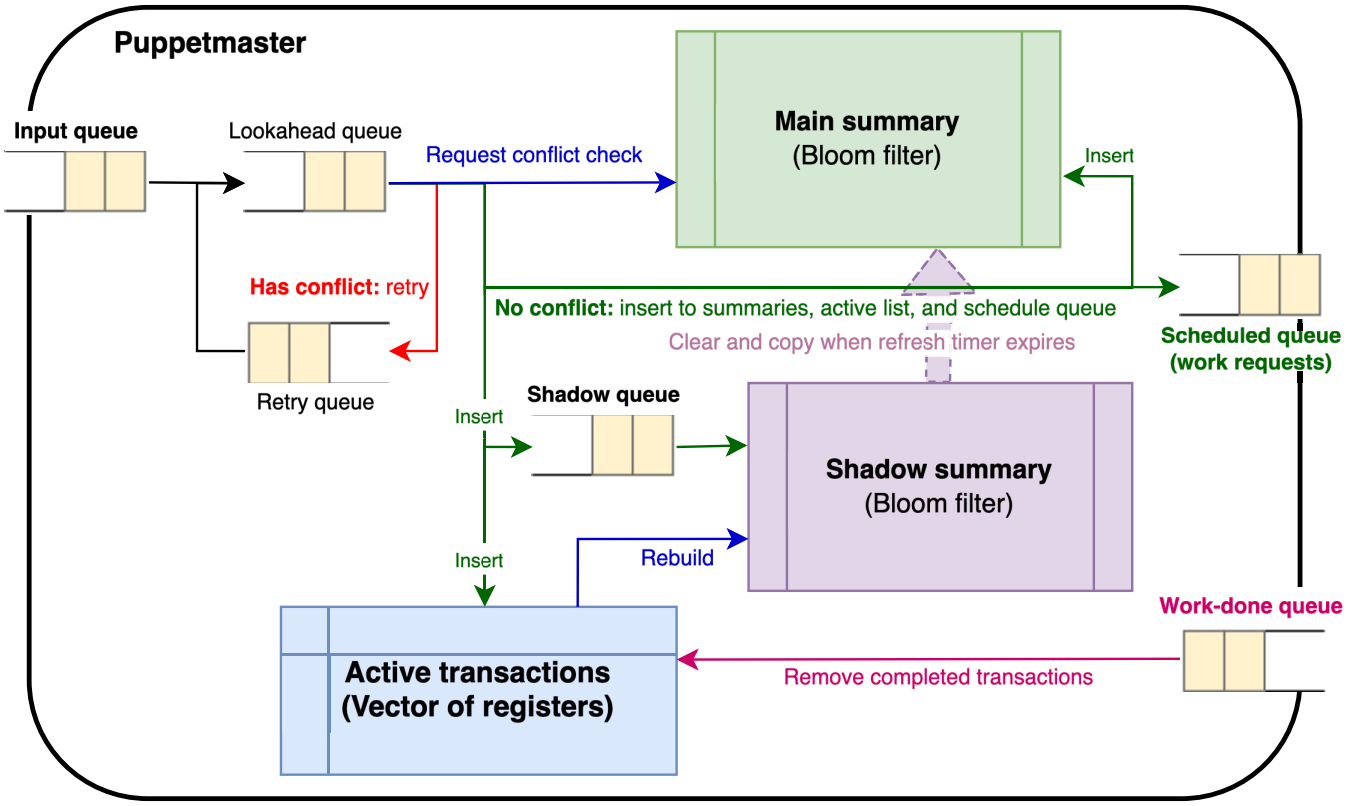

Figure 3.3: Puppetmaster's internal architecture

3.6.1 Input pipeline and lookahead logic

Transactions arrive into Puppetmaster through an input queue, which may be populated by either the software interface or an internal hardware transaction generator. To implement a bounded lookahead mechanism—similar to the one used in software—we stage incoming transactions through two additional buffers: a small *lookahead queue*, and a *retry queue* used for rejected transactions.

An arbiter selects between the main input queue and the retry queue when feeding the lookahead queue. Prioritizing the retry queue ensures that transactions that were previously rejected due to conflicts or lack of scheduling capacity are reconsidered promptly. Transactions in the lookahead queue are examined one-at-a-time. This architecture ensures

that the scheduler has a short but flexible window of pending transactions to consider without stalling on every rejection.

3.6.2 Conflict detection and eventual removal

Each transaction in the lookahead queue is checked for conflicts against the currently active transactions, conservatively tracked using Bloom filters. We refer to this data structure for conflict detection as a *summary*. Designing the Bloom filter involves various parameters, such as whether to use combined or separate filters for read and write sets, as well as decisions about sizing—balancing false positives against the FPGA logic resource constraints. Abstracting these implementation choices behind the summary interface allows more effective exploration of the design space. Specific details of our Bloom-filter implementation are provided in section 3.7.

Because Bloom filters do not support deletions, Puppetmaster cannot individually remove transaction entries. Instead, it maintains a pair of Bloom filters: a *current filter*, tracking active transactions; and a *shadow filter*, accumulating all scheduled transactions since the last refresh. Transactions are inserted into both filters simultaneously. Periodically, the current filter is replaced by the shadow filter, and the shadow filter is cleared. After a refresh cycle, the shadow filter is rebuilt in the background by traversing the list of active transactions.

While this approach conservatively tracks transactions—potentially including completed ones—it guarantees conflict-safety and effectively exploits hardware parallelism. By amortizing the refresh operation over multiple transactions, Puppetmaster avoids complex per-transaction cleanup logic. Crucially, all transactions are *eventually* removed, preventing long-term accumulation of stale state.

3.6.3 Adaptive refresh strategies

In the current implementation, the shadow filter is copied to the main filter after a fixed number of conflict-check failures. While simple, this strategy can waste cycles: during a copy, all conflict checks must pause. If no transactions were actually removed from the active list, the copy is unnecessary and prevents conflict checks from running in a tight loop. In dense or high-conflict workloads, stale entries may linger too long, increasing the false-positive rate; refreshing earlier could improve throughput.

A more adaptive approach would monitor the difference between the number of *currently active transactions* (i.e., those not yet reported as complete) and the number of *transactions represented in the Bloom filter*. The latter can be approximated using a counter that tracks how many transactions have been inserted into the filters since the last refresh. When the gap between these two counts grows large, it suggests that many completed transactions are still inflating the filter's state and contributing to false positives. Conversely, when the gap is small, the filter is still fresh, and refreshes can be deferred. This heuristic provides a lightweight and general-purpose mechanism to improve scheduling efficiency without needing deep inspection of the filter state.

3.7 Bloom filter sizing and synthesis results

Earlier, we introduced the *summary* abstraction for conservative conflict detection. In this section, we detail our implementation using Bloom filters, focusing on the critical design considerations and trade-offs. Specifically, we selected filter parameters that balance false-positive rates, FPGA resource usage, and scheduling latency.

3.7.1 Chunked Bloom-filter architecture

The primary design goal of our Bloom filter is to minimize false-positive conflict detections. To this end, we adopt a parallel Bloom filter architecture with k partitions, each corresponding to a different hash function. This approach reduces false positives compared to single-hash filters by independently checking object membership across multiple hash spaces. A detailed probabilistic analysis is provided in Appendix D.

However, directly checking all bits for every object in a large transaction is impractical. A single transaction may contain up to 16 objects (8 reads and 8 writes), requiring 64 bit probes when using 4 hash functions. Probing all of these in parallel would require excessive combinational logic and consume an impractical amount of LUTs.

We consider two main strategies to reduce resource usage.

The first strategy is to process one object at a time. For each object, we fetch one chunk from each partition and check the corresponding bits. This strategy reduces logic requirements but introduces latency proportional to the number of objects per transaction.

The second strategy is to use a *chunked Bloom filter* architecture. Here, we subdivide each partition into c smaller *chunks*. We compare all objects in parallel against one chunk at a time. Each object maintains state indicating whether its corresponding bit is found in each partition. After scanning all chunks, we determine if any object matches all k bits—signaling a conflict.

These approaches can be combined. For example, we may check a small group of objects at a time against either full partitions or small chunks to balance throughput and resource usage.

Our prototype implements the chunked Bloom-filter design with the following parameters:

- $k = 4$ hash functions (partitions)
- $c = 8$ chunks per partition
- 256 bits per chunk

This set of parameters results in a filter of $m = 4 \times 8 \times 256 = 8192$ bits. The design enables narrow per-cycle logic, efficient BRAM utilization, and manageable LUT usage.

A partitioned Bloom filter of this size can store 778 objects or 48 transactions (assuming 16 objects per transaction) and have a false positive rate of 1%.

3.7.2 Area consumption by Bloom filters

To examine trade-offs between filter size, accuracy, and resource utilization, we evaluated several Bloom-filter configurations. Table 3.3 presents the total filter size, number of objects where false positive rate (FPR) is 0.1% and 1%, and FPGA-synthesis results for each configuration. The Bloom filters can insert or check for conflicts one transaction (8 reads and

8 writes) at a time. We synthesized the design, targeting AMD Virtex Ultrascale XCVU095 FPGA on the VCU108 development board, at the clock frequency of 125 MHz. The FPGA has 1,176,000 system logic cells, which translate to 537,000 LUTs and 1,075,200 flip-flops.

Configuration	m	**FPR=0.1%**	**FPR=1%**	**LUTs**	**FFs**	**WNS (ns)**
$4 \times 4 \times 256$	4096	200	389	77,620	4,285	0.068
$4 \times 4 \times 512$	8192	400	778	141,589	6,684	Failed
$4 \times 8 \times 256$	8192	400	778	75,081	4,343	0.068

Table 3.3: Estimated false positive rates and synthesis results for various Bloom filter configurations (partitions $\times$ chunks $\times$ bits per chunk)
LUTs stand for lookup tables. FFs stand for flip-flops. WNS stands for worst negative slack. Failed means synthesis tool cannot close timing.

Somewhat unintuitively, the $4 \times 8 \times 256$ configuration in the first row consumed more LUTs and flip-flops than the $4 \times 4 \times 256$ variant in the third row, despite having twice the bit count. We attribute this discrepancy to synthesis heuristics and backend nondeterminism. Notably, the $4 \times 4 \times 256$ design achieved better timing slack, suggesting that the synthesis tool had to exert less effort optimizing its logic overall. Additionally, the overall LUT and FF usage is higher than expected across all configurations. We suspect this may be due to suboptimal static elaboration during Bluespec-to-Verilog translation, which we aim to address in future revisions.

3.7.3 Separating read sets and write sets

Currently, a single shared Bloom filter tracks both reads and writes, simplifying the hardware design but potentially causing higher false positives in read-intensive scenarios. Supporting distinct Bloom filters for read and write sets would significantly enhance scheduling precision at the cost of double the memory size. Our summary abstraction easily accommodates such optimizations, enabling straightforward exploration of future architectural improvements.

Chapter 4

Evaluating Puppetmaster

Is it worth accelerating transaction scheduling in hardware, given the overheads of software-hardware communication?

This chapter aims to answer that question through a sequence of focused experiments. We begin by establishing a software baseline. Using a fully software greedy scheduling pipeline discussed in section 3.5, we analyze both peak throughput and minimum achievable latency.

Then, we shift our attention to the hardware implementation of Puppetmaster. By benchmarking with the in-hardware transaction generator and executor, we assess the scheduler's throughput and latency in isolation. These results demonstrate that Puppetmaster performs well as a drop-in IP core for hardware-only scenarios and does not become a bottleneck in hybrid software-hardware systems.

Throughout this chapter, we test our hypothesis: *that transaction scheduling is a suitable candidate for hardware acceleration, even in the presence of moderate communication overheads.*

4.1 YCSB Workload

To evaluate Puppetmaster's scheduling performance across varying contention levels, we use the Yahoo! Cloud Serving Benchmark (YCSB) [5]. YCSB is widely used to model key-value store workloads and offers tunable parameters that allow us to control access skew and write intensity.

Each benchmark run uses a single transaction type that accesses a fixed number of records (typically 16), randomly drawn from a database of size $N = 20,000,000$. Record access follows a Zipfian distribution with skew parameter θ. Each access has a write probability ω.

We adopt $\theta = 0$, 0.6, and 0.8 to model low, medium, and high contention, consistent with [23]. We focus on two YCSB workload classes: read-heavy ($\omega = 0.05$) and write-heavy ($\omega = 0.5$).

Figure 4.1 shows the fraction of accesses targeting the top 10% of records for different θ values. As θ increases, contention increases significantly: at $\theta = 0.8$, 60% of accesses go to just 10% of the records.

We assume a fixed execution time per scheduled transaction, ignoring caching or locality effects. Since our focus is on scheduling, not data layout, we treat the record structure as

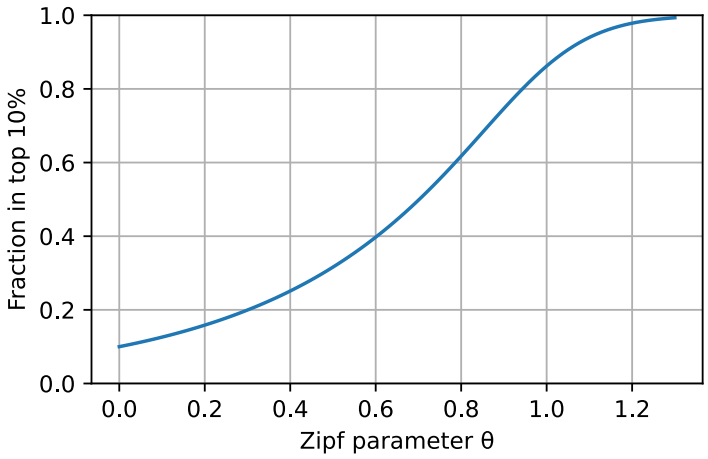

Figure 4.1: Fraction of accesses targeting top 10% of the records with Zipfian distribution

opaque. All transactions are independent and uniform in type, simplifying the analysis of scheduler throughput and latency.

4.2 Software Evaluation

We begin by characterizing the performance of Puppetmaster's software-only implementation. All experiments in this section use 8 puppets and synthetic workloads generated offline. Transactions vary in object count, access pattern skew (Zipf θ), and read/write ratio.

Experiments are conducted on a consumer laptop with a 6-core (12-thread) AMD Ryzen 5 7640U processor. To reduce measurement noise, we prioritize the testbench process using `chrt` and report only the best-performing run (i.e., least affected by interference).

Other relevant configuration parameters are as follows:

- The system supports up to 16 puppets (as defined at compile time).
- Each transaction may access up to 16 objects.
- Each puppet can have up to 8 concurrently scheduled transactions, limiting both its output queue and work-completion queue sizes.
- The client maintains a queue of up to 64 pending transactions.

To reduce logging overhead, only one out of every 2^{14} transactions is recorded.

4.2.1 Best-Case Throughput and latency

To establish a baseline for comparison, we evaluate Puppetmaster under idealized conditions. We use a workload where each transaction accesses zero objects—resulting in no conflicts—and performs no actual work (0 μs duration). The workload contains 100 million transactions.

To improve histogram visibility, we exclude the top 2% of latency values, which are more likely attributed to noise (e.g., logging or progress checks).

The system achieves a steady-state throughput of over 11 million transactions per second. The corresponding end-to-end latency (from submission to work done) distribution is presented in Figure 4.2, showing a mean latency of roughly 6 μs.

We note that varying the number of puppets should not, and indeed does not, affect the result in this best-case simulation.

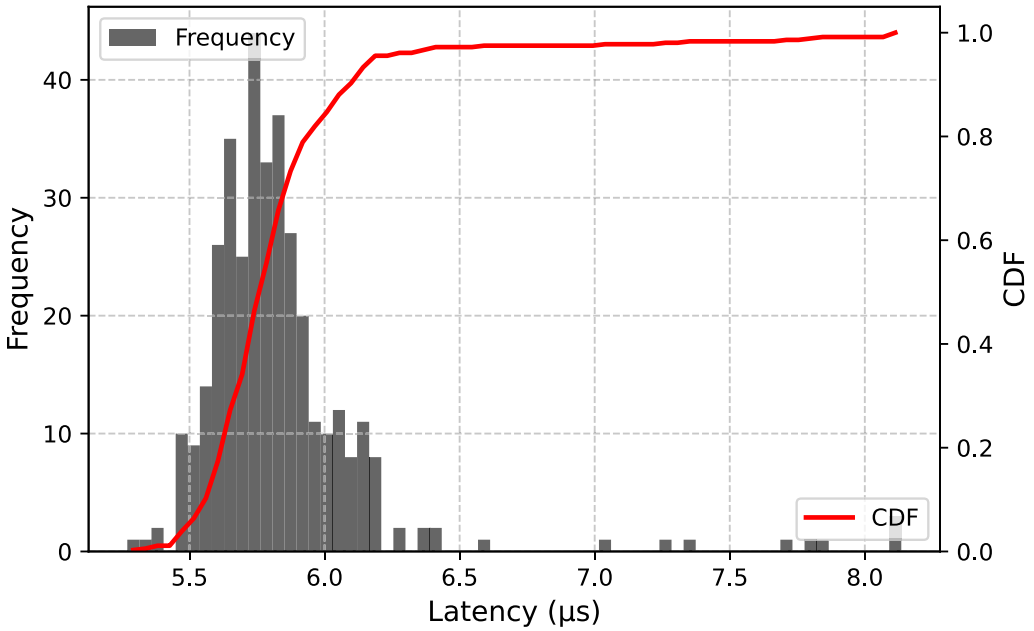

Figure 4.2: End-to-end latency for zero-object workload (unthrottled)

However, this latency is not representative of intrinsic scheduling cost. Because the client submits transactions faster than the scheduler can process them, most transactions queue up and incur queuing delay. To obtain a more meaningful latency measurement, we throttle the client to submit at no more than half the maximum throughput.

The throttled run yields a much lower mean latency—around 330 ns—shown in Figure 4.3. Minor histogram gaps and bimodal artifacts are due to timing granularity and logging effects.

4.2.2 Throughput and latency under load

We evaluate the system under various YCSB workloads introduced in section 4.1. Summary results are presented in Table 4.1, and we make several key observations:

- Transaction throughput is most sensitive to the number of objects per transaction. In contrast, access type (read vs. write) and skew level have comparatively minor effects on overall throughput and latency.
- Somewhat counterintuitively, higher contention (i.e., skewed access patterns) often results in lower latency. A plausible explanation is improved branch-prediction behavior in the scheduler under more predictable access patterns.
- Thread communication and queue multiplexing introduce approximately 150–200 ns of latency. This latency is most pronounced in the zero-object (throttled) workload, where submission-to-receipt and cleanup stages dominate the timing.
- The scheduling logic itself—responsible for conflict detection and puppet assignment—contributes roughly 100–200 ns of latency, depending on the transaction size.

Run	Workload	Throughput (txn/s)	E2E Latency (mean, μs)	Stage Latency (mean, μs)			
				Sched.	Recv.	Done	Cleanup
1	Zero-object	11,186,520.38	5.84	5.58	0.148	0.825	2.67
2	Zero-object (throttled)	–	0.332	0.146	0.134	0.049	0.174
3	8 objs, read, low, 5 μs	1,569,509.55	59.87	40.45	14.15	5.11	1.13
4	8 objs, read, medium, 5 μs	1,570,282.01	68.73	40.30	23.11	5.11	1.35
5	8 objs, read, high, 5 μs	1,552,408.34	63.61	39.82	18.07	5.12	2.46
6	8 objs, write, low, 5 μs	1,112,837.28	62.92	56.81	0.80	5.12	26.16
7	8 objs, write, medium, 5 μs	1,088,430.28	64.23	58.32	0.67	5.12	27.20
8	8 objs, write, high, 5 μs	1,328,538.51	54.10	47.63	1.19	5.12	6.28
9	8 objs, read, low, 5 μs (throttled)	–	5.53	0.255	0.196	5.07	0.177
10	8 objs, read, medium, 5 μs (throttled)	–	5.54	0.238	0.221	5.07	0.174
11	8 objs, read, high, 5 μs (throttled)	–	5.51	0.213	0.216	5.07	0.180
12	8 objs, write, low, 5 μs (throttled)	–	5.62	0.258	0.253	5.06	0.192
13	8 objs, write, medium, 5 μs (throttled)	–	5.84	0.231	0.208	5.06	0.181
14	8 objs, write, high, 5 μs (throttled)	–	5.54	0.258	0.213	5.06	0.181
15	16 objs, read, low, 20 μs	385,716.15	283.65	160.51	102.87	20.13	0.203
16	16 objs, read, medium, 20 μs	385,237.84	293.37	150.90	112.99	20.12	0.620
17	16 objs, read, high, 20 μs	371,471.31	248.22	166.91	58.11	20.12	1.03
18	16 objs, write, low, 20 μs	383,439.85	248.02	159.45	68.13	20.12	21.93
19	16 objs, write, medium, 20 μs	368,277.32	202.73	169.42	12.43	20.12	54.55
20	16 objs, write, high, 20 μs	289,154.28	228.52	203.20	4.80	20.12	1.94
21	16 objs, read, low, 20 μs (throttled)	–	20.65	0.401	0.180	20.06	0.175
22	16 objs, read, medium, 20 μs (throttled)	–	20.69	0.446	0.170	20.07	0.175
23	16 objs, read, high, 20 μs (throttled)	–	20.72	0.321	0.316	20.07	0.200
24	16 objs, write, low, 20 μs (throttled)	–	20.62	0.381	0.185	20.06	0.171
25	16 objs, write, medium, 20 μs (throttled)	–	20.60	0.321	0.205	20.07	0.185
26	16 objs, write, high, 20 μs (throttled)	–	20.65	0.346	0.170	20.06	0.175

Table 4.1: Performance metrics across software test runs with varying configurations.

Workload names follow the pattern: number of objects per transaction, YCSB workload type (read-heavy or write-heavy), access skew (low, medium, high), and target execution time per transaction (e.g., 5 μs).
Sched. is the time from transaction submission to when it is scheduled by Puppetmaster.
Recv. is the time from scheduling to when the responsible puppet receives the transaction.
Done is the time from receipt to execution completion.
Cleanup is the time from completion to removal from Puppetmaster's active set.

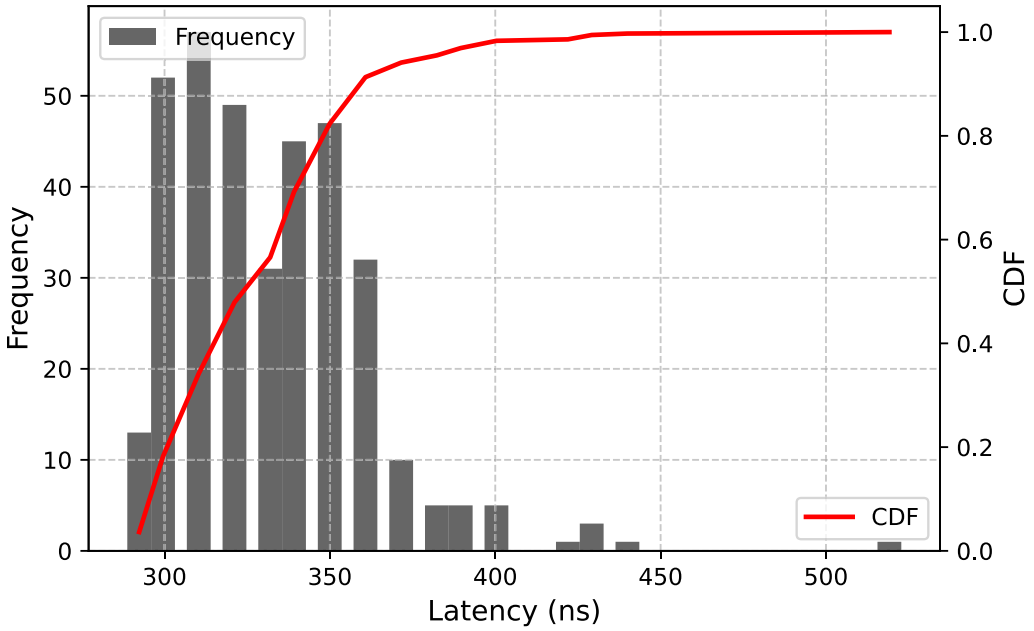

Figure 4.3: End-to-end latency for zero-object workload (throttled)

We note that our software implementation is not optimized for maximum absolute throughput. However, we believe the performance is representative. Informal testing using the DBx1000 system from [23] on the same machine indicates that roughly half of the additional latency and throughput loss stems from concurrency-control logic alone (verified by substituting in a dummy concurrency-control mechanism).

Overall, our system introduces approximately 0.5–0.7 μs of latency overhead per transaction, excluding the user-specified execution time. We achieve 80–90% of the theoretical maximum throughput under low-to-moderate contention levels.

Crucially, if communication overheads are minimized—e.g., using PCIe with DMA rings—the scheduler has roughly 100–200 ns per transaction to make a scheduling decision. On an FPGA, this translates to tens of clock cycles, reinforcing the feasibility of implementing this logic in hardware without becoming the bottleneck.

4.3 Hardware Evaluation

We now evaluate Puppetmaster's hardware implementation, targeting the architecture described in section 3.6. The design was successfully synthesized on a Xilinx VCU108 development board, targeting a 125 MHz clock with positive timing slack. The scheduler was implemented in Bluespec and integrated with our testbench using Connectal.

4.3.1 Synthesis Results

Table 4.2 summarizes synthesis results for our default Bloom-filter configuration. The system supports up to 8 reads and 8 writes per transaction, and tracks a bounded list of up to 64

active transactions at a time. The design fits comfortably within FPGA resource budgets and is timing-clean at the target frequency.

Configuration	Max Objs	LUTs	FFs	Freq
$4 \times 4 \times 256$ (combined RW)	8R+8W	173,836	37,278	125 MHz

Table 4.2: Synthesis results for the default hardware-scheduler configuration.

4.3.2 Performance results

The hardware design is synthesized for exactly 8 read and 8 write objects per transaction, with the Bloom filter tracking the combined set. As a result, we use only a subset of the workloads from the software evaluation. To accommodate the slower runtime of Verilog simulation, each run contains just 1,000 transactions. However, the use of simulation enables precise, cycle-accurate timing measurements without sampling noise. All transactions are logged, and no outliers are removed.

Results are summarized in Table 4.3.

Run	Workload	Throughput (txn/s)	E2E Latency (mean, μs)	Stage Latency (mean, μs)			
				Sched.	Recv.	Done	Cleanup
1	Zero-object	3,906,860.45	16.20	16.18	0.008	0.008	0.008
2	8R+8W, low contention, 5 μs	1,521,829.12	27.42	13.98	8.45	5.01	0.008
3	8R+8W, medium contention, 5 μs	1,544,888.27	27.37	13.74	8.62	5.01	0.008
4	8R+8W, high contention, 5 μs	1,507,022.73	27.29	14.11	8.18	5.01	0.008
5	8R+8W, low contention, 10 μs	792,342.80	56.33	26.58	19.47	10.01	0.008
6	8R+8W, medium contention, 10 μs	792,212.80	56.31	26.82	19.42	10.01	0.008
7	8R+8W, high contention, 10 μs	791,464.84	56.34	26.83	19.46	10.01	0.008

Table 4.3: Performance metrics across hardware test runs with varying configurations

Workload names follow the pattern: number of read and write objects per transaction, access skew (low, medium, high), and target execution time per transaction (e.g., 5 μs or 625 cycles).
Sched. is the time from transaction submission to when it is scheduled by Puppetmaster.
Recv. is the time from scheduling to when the responsible puppet receives the transaction.
Done is the time from receipt to execution completion.
Cleanup is the time from completion to removal from Puppetmaster's active set.

Across all workloads, the scheduler achieves over 90% of the theoretical maximum throughput, even at the maximum transaction size (16 objects). This level of performance is a strong result, as software schedulers typically degrade significantly at this scale. Synthesizing for specific workloads—e.g., with fewer objects or finer-grained conflict tracking—could yield even better performance.

The scheduler's critical path is approximately 17–19 cycles (144 ns). This number includes 8 cycles for conflict checking and 8 for insertion into the shadow summary. Although insertion can overlap with dispatch, it stalls subsequent checks. This latency can be reduced by

enabling concurrent check-insert operations, e.g., via atomic pointer swaps instead of copying during refresh.

We also modeled a constant communication latency and confirmed that latency-hiding techniques (e.g., queueing) were effective—throughput remained stable.

Overall, these results show that Puppetmaster's hardware scheduler provides low-latency, high-throughput scheduling. With an ASIC implementation at higher clock speeds, we expect performance improvements of 5×–10×, making Puppetmaster a compelling drop-in IP core for transaction-heavy systems.

4.3.3 Summary

Despite the added complexity of hardware design and synthesis, Puppetmaster achieves strong performance metrics across a range of workloads. The scheduler's latency and throughput are competitive with a highly optimized software baseline, even before considering the benefits of hardware parallelism or faster fabrication technologies. These results support our central hypothesis and motivate the discussion of broader implications in the next chapter.

Chapter 5

Discussion and future work

5.1 Discussion

Our evaluation demonstrates that the hardware scheduler presented in this thesis can reliably handle transaction scheduling at a high throughput, even under highly contended workloads. This performance result strongly supports the viability of hardware acceleration for transaction scheduling, especially considering the significant overhead incurred by concurrency-control mechanisms in current database systems. Given modern databases allocate roughly half of their CPU cycles to concurrency control, offloading scheduling tasks onto dedicated hardware holds substantial promise for enhancing overall system performance.

A key advantage of our FPGA-based approach lies in the inherent customization and parallelism that it provides. Unlike software solutions, where minor concurrency improvements often do not justify dedicating CPU cores, FPGAs enable the exploitation of small concurrency opportunities at negligible additional cost. This characteristic makes hardware acceleration particularly attractive for concurrency control, potentially unlocking performance improvements that are impractical to achieve in software-only implementations.

Additionally, in purely hardware-centric workloads, Puppetmaster's drop-in IP core simplifies and accelerates system prototyping and development. Although our current implementation provides robust performance, there is significant scope for further optimization tailored to specific FPGA platforms, such as those provided by Xilinx. These targeted improvements could yield additional performance benefits, enhancing Puppetmaster's attractiveness for specialized hardware applications.

5.2 Future work

5.2.1 Improved hashing scheme

Our current implementation employs a relatively simple FNV-1-style hash function due to our benchmark's randomized workloads lacking locality. While adequate for our initial testing scenarios, this hash function's simplicity may limit effectiveness in workloads exhibiting greater locality or more complex access patterns. Incorporating advanced hash functions, such as multiply-shift schemes [6], could reduce conflict-detection inaccuracies. However, these

advanced hashing methods would require careful pipelining of the preprocessing stages on the input queues to maintain throughput efficiency. Nonetheless, since our conflict-checking pipeline already introduces some inherent latency, preprocessing can be integrated without affecting overall throughput, effectively masking additional processing overhead.

5.2.2 Improved communication drivers

Currently, our software-hardware interface relies on Connectal's portal-based communication, incurring substantial latency and limiting throughput due to kernel involvement per transaction. A potential future improvement involves adopting a custom driver utilizing lock-free DMA ring buffers that allow kernel-free communication. The lock-free DMA-based communication approach, inspired by FaRM (Fast Remote Memory) [7], could significantly reduce communication latency and overhead, further enhancing the benefits of hardware acceleration.

5.2.3 Alternative deployment platforms

Exploring other FPGA deployment platforms could enhance Puppetmaster's practical adoption. Specifically, system-on-chip (SoC) platforms such as AMD Zynq offer lower latency interactions between CPUs and FPGAs, beneficial for applications compatible with embedded-scale computing. Additionally, broader testing on widely accessible FPGA platforms, including educational development boards and cloud-based offerings such as Amazon EC2 F1 and F2 instances, could increase accessibility and facilitate broader experimentation and adoption.

5.2.4 Revisiting original Puppetmaster architecture

While we have set aside the initial Puppetmaster architecture discussed in Appendix C due to FPGA area constraints, we believe the original approach still holds potential. Identifying compelling use cases and demonstrating an efficient end-to-end system using the original design remains a significant objective. The modular nature of Puppetmaster's design, with well-defined interfaces, will facilitate future experimentation in this area.

5.2.5 Support for dependent transactions

Our current implementation supports only static transactions (i.e., transactions with predefined read and write sets). Extending Puppetmaster to handle dependent transactions—where read/write sets are not predetermined—can be achieved using optimistic lock location prediction (OLLP), a technique introduced in [21]. Supporting dependent transactions would enable evaluations using more sophisticated benchmarks, such as TPC-C and STAMP, representing a valuable advancement in our research.

5.2.6 Comparative evaluations with related work

Comparing Puppetmaster more rigorously with related systems like ROCoCoTM [15] would further highlight our system's unique strengths and limitations. Unlike ROCoCoTM, which im-

plements optimistic concurrency control and explicitly targets CPU execution, Puppetmaster employs pessimistic concurrency control and remains execution-platform agnostic.

In contrast to traditional software-based concurrency-control methods such as two-phase locking, timestamp ordering, and multi-version concurrency control, which typically manage conflicts by stalling or aborting transactions reactively, Puppetmaster employs proactive job scheduling. By dynamically selecting and reordering independent transactions to avoid conflicts upfront, Puppetmaster can achieve greater parallelism and reduce runtime overhead from contention, especially under moderate-to-high conflict scenarios.

Future work should include direct performance comparisons against both hardware-accelerated methods like ROCoCoTM and conventional concurrency control-schemes. Such comparative evaluations will clarify the performance benefits of Puppetmaster's proactive scheduling approach and further demonstrate its suitability for diverse workloads and hardware configurations.

5.2.7 Cache-awareness and core-assignment optimization

Incorporating cache-awareness into transaction-assignment decisions presents another promising optimization. Rather than assigning transactions arbitrarily, Puppetmaster could maintain information regarding core data locality, assigning transactions to cores that minimize data movement and enhance cache performance. Because the core-assignment problem is largely orthogonal to conflict detection, this improvement could be developed independently.

Appendix A

Source code and documentation

All artifacts produced during the development of this thesis are available at:

https://tcpc.me/meng-thesis

Specifically, the repository includes source code for all software and hardware implementations, as well as all data analysis and plotting scripts. For educational purposes, I have also included a collection of notes on various topics related to the project, which may be useful for students interested in continuing this line of research. These notes are included in the same repository.

Appendix B

Review of single-producer single-consumer (SPSC) queues

Traditional queue implementations often rely on locking mechanisms (e.g., mutexes) to prevent concurrent modifications. However, locking introduces significant overhead, especially in performance-critical applications. Lock-free single-producer single-consumer (SPSC) queues eliminate locking overhead by structuring concurrent access in a way that does not require explicit mutual exclusion.

The key idea behind lock-free SPSC queues is to decouple the producer and consumer operations completely. Specifically, the producer only needs to verify available space at the queue's tail before inserting elements, while the consumer independently accesses the front element (head) of the queue. This clear separation allows producer and consumer operations to proceed largely asynchronously.

To ensure correctness without explicit locks, atomic operations are employed. Atomics in this context primarily serve as memory-ordering tools rather than for mutual exclusion. For instance, the producer must ensure data is fully written into the buffer before incrementing the tail index; similarly, the consumer must fully read data before incrementing the head index. While atomics conveniently manage memory ordering, explicit memory fences could achieve similar guarantees.

A correct implementation uses specific atomic memory orders. On the producer side, the tail index must be loaded immediately using *acquire* semantics to confirm available space, while the head can be loaded with *relaxed* semantics as it only indicates previously consumed slots. Conversely, on the consumer side, the head index must be stored using *release* semantics to indicate completed reads, and the tail can be loaded relaxed.

Caching head and tail indices locally on the producer and consumer sides can further reduce atomic overhead. On the producer side, caching the consumer's head index is beneficial when the queue is rarely full, reducing redundant atomic reads. Conversely, on the consumer side, caching the producer's tail index is useful when the queue is rarely empty. In scenarios where the queue is frequently full or empty, caching may offer minimal benefits or even introduce overhead due to stale indices.

This SPSC design does not generalize to multiple producers or multiple consumers. For example, if two producers simultaneously increment the tail, one may overwrite the other's data or skip buffer slots; likewise, concurrent consumers may both attempt to read or advance

the head, leading to lost or duplicated elements. General-purpose MPSC, SPMC, or MPMC queues require additional synchronization primitives or patterns—typically involving atomic compare-and-swap (CAS) operations or ticket-based schemes—to coordinate access among multiple threads. As a practical stop-gap, one effective way to simulate an MPSC queue (when precise size bounding is not essential) is to allocate one SPSC queue per producer and have the consumer time-multiplex between them. This strategy avoids contention between producers and can be surprisingly performant; Puppetmaster's software simulation adopts this strategy. Alternatively, producers may use a mutex to guard access to a shared queue, though this approach inevitably reintroduces the locking overhead that lock-free designs are meant to avoid.

Appendix C

Review of the prior Puppetmaster architecture

The following appendix discusses the previous design of Puppetmaster [19]. It has been superseded by the one in chapter 3. We include this review, supplemented with commentary, to shed light on the challenges inherent to this research area.

C.1 Prior algorithm overview

Puppetmaster maintains a list of all transactions it has received. A transaction is retained in memory until Puppetmaster no longer requires information about it because it has completed execution. We refer to the currently stored transactions as "live" transactions. An object is considered "live" if it appears in the read or write set of any live transaction.

To reduce resource usage across the architecture, Puppetmaster maintains a global "renaming table" for live objects. This table maps relevant 64-bit object identifiers into a much smaller internal object name space. Once a transaction is freed, its object names may be reused. However, at any given time, each name may represent at most one live object. A typical object name is 10 bits in length, allowing representation of up to 1024 live objects.

When a transaction descriptor arrives, Puppetmaster first processes it through the "renamer" to produce a compact representation of its read and write sets. The renamed transaction is then placed into the "scheduling pool," a buffer where it awaits execution. Scheduling is triggered when this buffer reaches capacity. A typical scheduling pool contains up to 128 transactions.

Once the scheduling pool is full, Puppetmaster runs the "tournament-scheduling" algorithm to determine a subset of transactions that can be executed in parallel. These transactions are marked as "started" and sent to the executor. The remaining transactions stay in the scheduling pool. This process is repeated whenever a new scheduling pool is formed, with Puppetmaster ensuring that no new transactions conflict with those currently in execution. Upon completion, the executor notifies Puppetmaster, which can then safely free the associated transaction metadata. The overall dataflow is illustrated in Figure C.1.

The tournament-scheduling algorithm proceeds in multiple rounds. In each round, transactions are paired together. If two transactions do not conflict, they are merged into a single

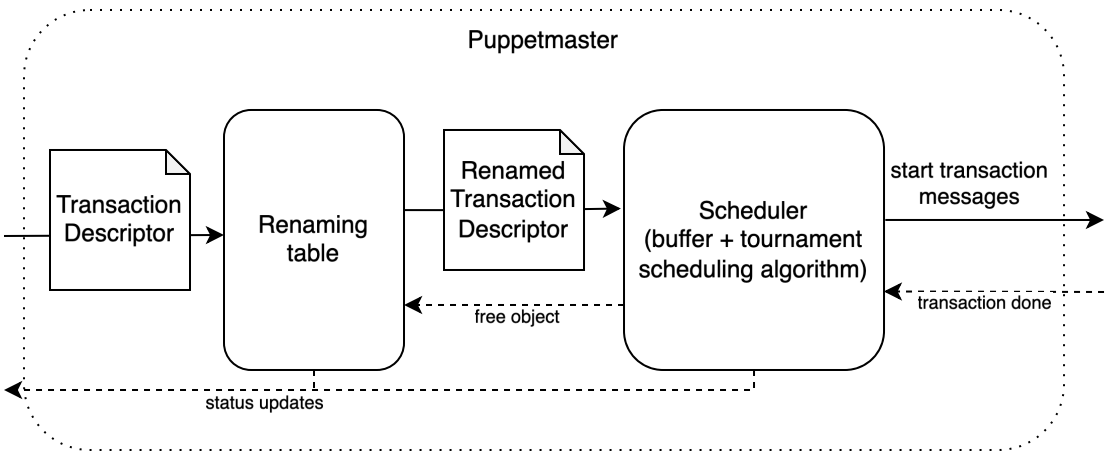

Figure C.1: Flow of transaction data through Puppetmaster

composite transaction. If they do conflict, one transaction is chosen arbitrarily and the other is discarded. This results in halving the number of transactions in each round. The process continues until only one transaction remains. Puppetmaster retains metadata to reconstruct the original transactions from the merged result and issue correct start-work messages.

These transaction pairings can be processed in parallel, so the overall scheduling latency is logarithmic in the pool size, assuming full parallelism.

In hardware, a transaction is represented using an index bitset, a read bitset, and a write bitset. The index bitset indicates which transactions in the scheduling pool are represented by the current merged transaction. In the first round, the index bitset is simply a one-hot encoding of the transaction's position. The final output of the tournament-scheduling process is an index bitset representing the set of transactions to start.

The read and write bitsets serve similar roles, each representing the union of the read or write sets of the merged transactions. Conflicts between transactions can be checked efficiently using bitwise AND and OR operations.

The need for the renaming process is now evident. Representing read and write sets as bitsets enables fast conflict detection in hardware but requires a small object universe. Renaming compresses the 2^{64} possible object IDs into a more manageable space—e.g., 1024 objects. For a scheduling pool of 128 transactions, the total storage required for index, read, and write bitsets is at most $128 \cdot (128 + 1024 + 1024)$ bits, or approximately 272 kilobits.

If the scheduling pool is limited to a single transaction, the algorithm reduces to greedy scheduling, in which each transaction is evaluated sequentially for compatibility with currently executing ones. Conversely, if the scheduling pool encompasses the entire workload, the algorithm behaves as a heuristic for offline scheduling. However, such an approach would be impractical due to its high latency and resource requirements.

C.2 Evaluation of the tournament-scheduling scheme

This section investigates the characteristics of the tournament-scheduling algorithm, specifically how many transactions it can successfully identify as conflict-free from a fixed-size input

workload. We use the YCSB benchmark described in section 4.1 to perform a performance comparison between the tournament scheduler and a baseline greedy scheduler. We conclude the section by listing several factors that contribute to our decision to re-design the system.

We conduct simulations by first generating an address space corresponding to the database with N unique addresses. Larger address spaces imply lower contention. Next, we generate a workload consisting of P transactions ("pool size"), with fixed parameters θ (Zipf parameter) and ω (write probability). These transactions are processed through both greedy and tournament schedulers, recording the number of transactions successfully extracted. Each scenario is run 10 times, and results are averaged.

Results showing the number of transactions extracted as a function of database size are presented in Figure C.2, where we set $P = 128$. For clarity, we separate the read-heavy and write-heavy workloads into distinct plots and evaluate both small (8 records per transaction) and large (16 records per transaction) scenarios. Each plot shows six lines representing low, medium, and high contention for both greedy and tournament schedulers.

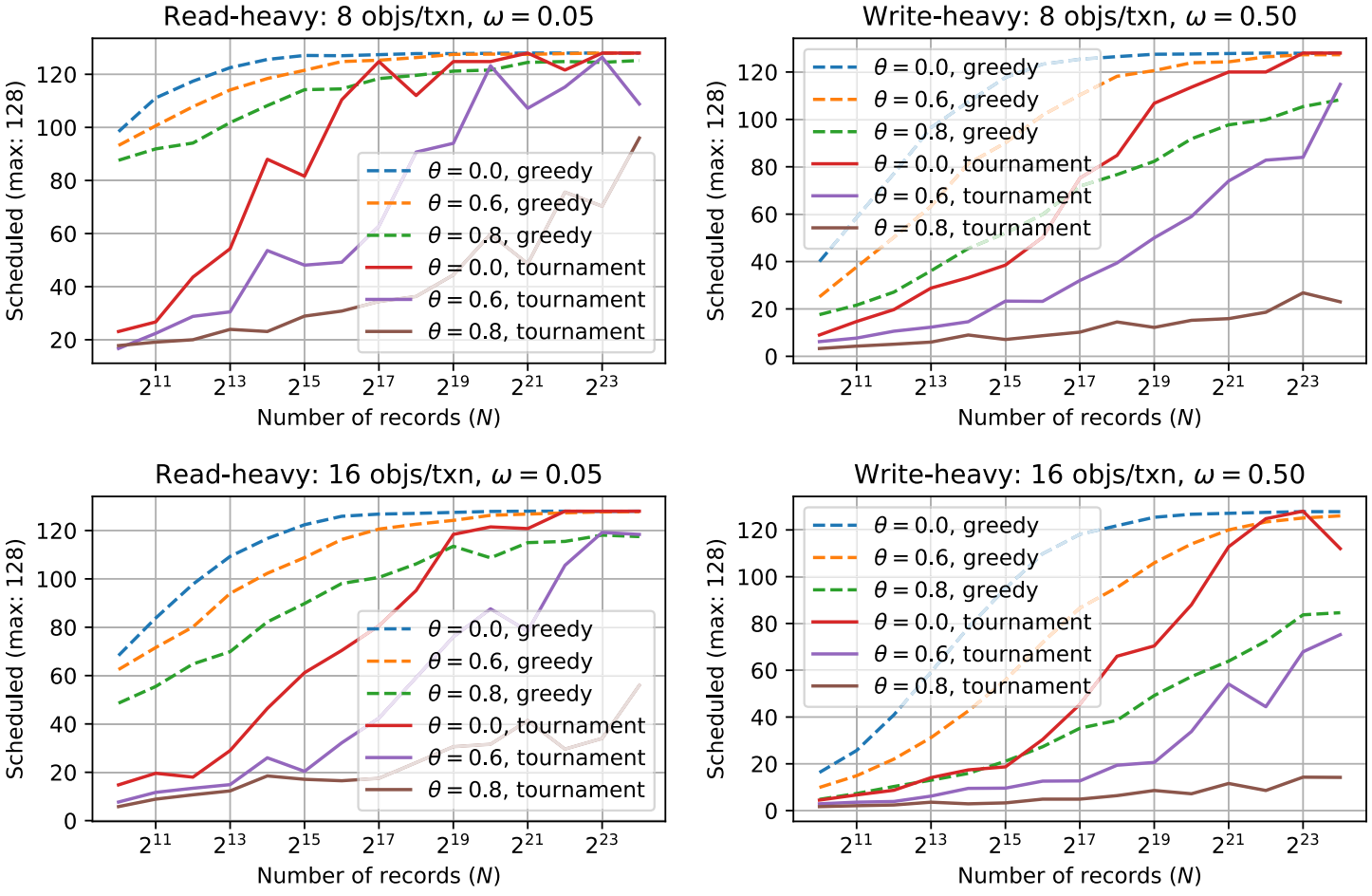

Figure C.2: Number of transactions extracted by the greedy scheduler versus the tournament scheduler in various workloads with $P = 128$

In general, the tournament scheduler under-performs the greedy scheduler when contention

is high (small address spaces). This performance gap arises because in the final round, two batches of $P/2$ transactions are ideally merged—provided they are conflict-free—into a single batch of P. However, any conflicts between batches reduce the final batch size significantly, often halving it. Due to the large cumulative read- and write-sets, conflicts are highly probable. In contrast, greedy scheduling checks transactions individually against the currently running set, reducing the likelihood and impact of conflicts.

However, with sufficiently large address spaces (low-contention scenarios), the tournament scheduler matches greedy scheduler performance. Such conditions are realistic in typical database applications with extensive address spaces.

Additionally, increasing the scheduling pool size P does not necessarily lead to much better performance from the tournament scheduler, as illustrated in Figure C.3 with $P = 4096$. (We graphed only the read-heavy and write-heavy workloads with 16 objects per transaction.) With larger pools, merging large batches conflict-free becomes increasingly improbable, limiting effective pool utilization. Conversely, the greedy scheduler continues scaling effectively.

This limitation of tournament scheduling may or may not be acceptable in practical settings. Real-world performance depends not solely on scheduling throughput but also scheduling speed—a metric favoring tournament scheduling when implemented in hardware.

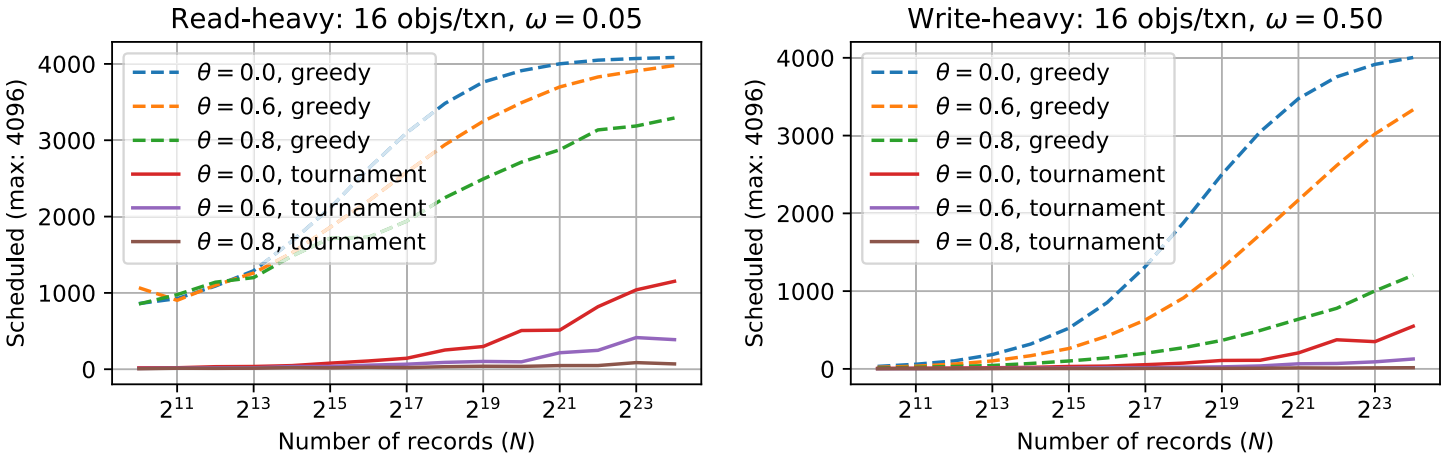

Figure C.3: Number of transactions extracted by the greedy scheduler versus the tournament scheduler in various workloads with $P = 4096$

Finally, we note that the tournament scheduler requires the scheduling pool to be full before it can schedule any transactions, so the latency generated by this scheme can be prohibitively large. The renaming process can also introduce additional latencies as well. Moreover, achieving performance comparable to a simple greedy scheduler requires a substantial FPGA area. Given that the rest of the architecture more-or-less already serializes the transaction stream, we believe a simpler greedy-algorithm-based architecture like in chapter 3 may be more appropriate.

Appendix D

Review of Bloom filters

Bloom filters [3] are space-efficient probabilistic data structures. Each Bloom filter represents a set S and supports the insertion of elements as well as membership testing. The membership test may return false positives but never false negatives.

A Bloom filter consists of a bit array of length m, initially all zeros, and a collection of k mutually independent hash functions $H_1, H_2, \ldots, H_k$. To insert an element x_i from the universe $\mathcal{U}$ into S, the bit positions $H_1(x_i), H_2(x_i), \ldots, H_k(x_i)$ are set to 1. To check whether q_i is in the set, the result is positive only if the bits $H_1(q_i), H_2(q_i), \ldots, H_k(q_i)$ are all set to 1.

If q_i is in the set, it will definitely return a positive result because the corresponding bits would have been set during insertion. If q_i is not in the set, it may still return a false positive if those bits were set by other elements. Otherwise, it correctly returns a negative result.

Two common variants are *unpartitioned* and *partitioned* Bloom filters. Unpartitioned Bloom filters use hash functions $H_i : \mathcal{U} \to [m]$, utilizing the entire m bits. Partitioned Bloom filters use hash functions of the form $H_i : \mathcal{U} \to \left[\frac{m}{k}\right]$, with each function's output offset by $i \cdot \frac{m}{k}$ to map into its corresponding partition.

D.1 False-positive analysis

Partitioned Bloom filters are typically preferred in hardware implementations, since the k partitions can be checked in parallel, with results combined using the AND operator. The approximate [8] false-positive probability is:

$$\left(1 - \left(1 - \frac{k}{m}\right)^{|S|}\right)^k,$$

where $|S|$ is the number of elements inserted.

Proof. Consider a membership test for $x_j \notin S$ in a single partition i. A negative result occurs if $H_i(x_j)$ points to a bit that is still 0. This negative result arises only when all previous $|S|$ elements hash to any of the other $\frac{m}{k} - 1$ positions, yielding a probability of $\left(1 - \frac{k}{m}\right)^{|S|}$. Therefore, the probability of a positive result in one partition is $1 - \left(1 - \frac{k}{m}\right)^{|S|}$. The final result is positive only if all k partitions return positives. □

The probability of false positives can be further approximated using a known limit:

$$\left(1 - e^{-\frac{k|S|}{m}}\right)^k.$$

We also empirically graphed the false-positive probability to illustrate its dependence on m, k, and $|S|$.

First, increasing the total number of bits m allows more elements to be inserted while keeping the false-positive rate low. Figure D.1 below shows a log-log plot of the false-positive rate versus the number of elements inserted ($|S|$), for $m = 512, 1024, 2048, 4096, 8192$ and $k = 4$, determined empirically. For example, with $m = 1024$, the theoretical false-positive rate is 2.4% for 128 elements and rises sharply to 16.0% at 256 elements. Thus, to keep the false-positive rate low, m must be large relative to $|S|$.

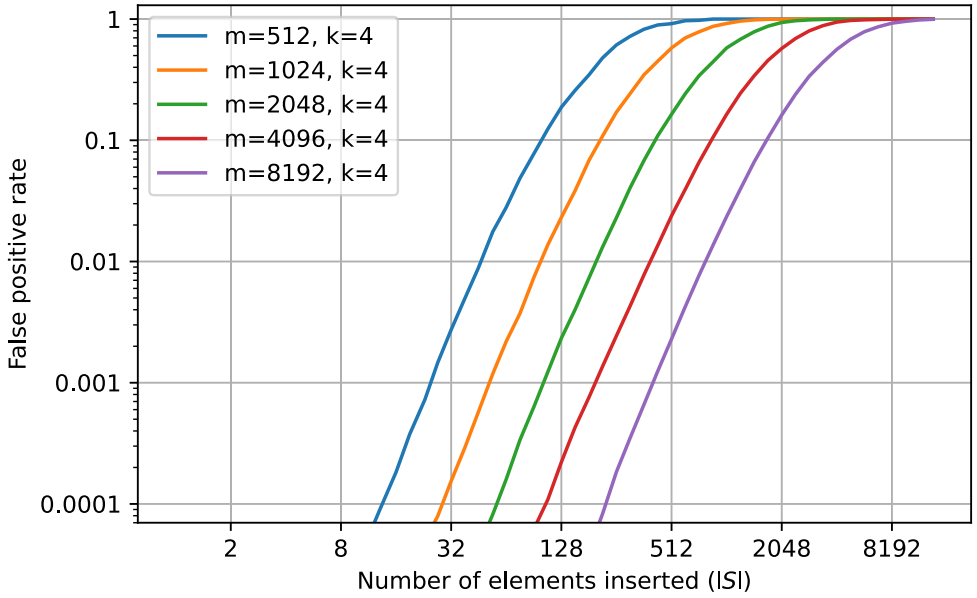

Figure D.1: False-positive rate of partitioned Bloom filters with varying m and fixed k

Second, if m is fixed and k varies, the rate of false positives changes, as shown in Figure D.2. However, the number of elements that can be supported before saturation remains roughly the same. For $k \ll m$, a higher k leads to lower false-positive rates before saturation. Note that k cannot grow arbitrarily. If $k = m$, even a single insertion could saturate the filter by setting all bits.

Finally, consider the case where m and k scale together, as shown in Figure D.3. As expected, more bits allow the Bloom filter to support more elements before saturation.

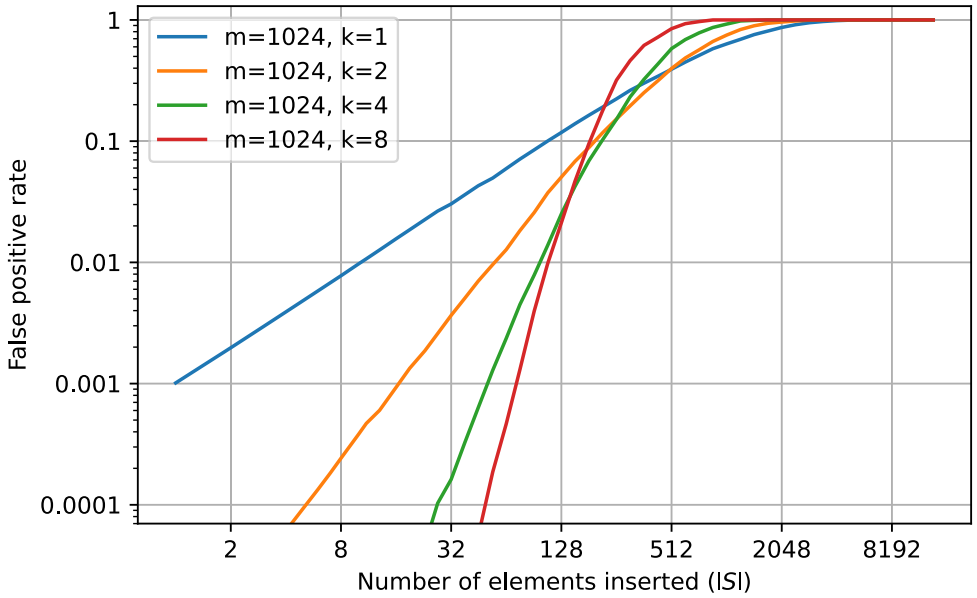

Figure D.2: False-positive rate of partitioned Bloom filters with fixed m and varying k

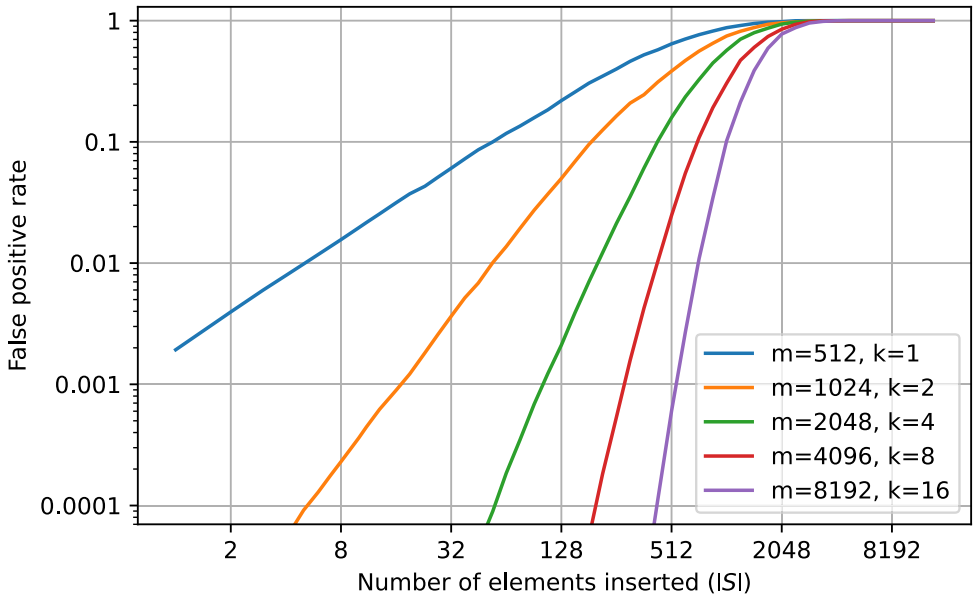

Figure D.3: False-positive rate of partitioned bloom filters with fixed m/k

D.2 Set-intersection Analysis

We now explore additional operations on Bloom filters: set intersection, union, and emptiness check. These are used to determine whether two transactions' read- and write-sets overlap.

Given two parallel Bloom filters with the same parameters and hash functions, their intersection can be approximated using a bitwise AND. This gives an approximate representation of the set intersection. However, the resulting Bloom filter has a higher false-positive probability than one built from the actual intersection of the raw sets.

For union, a bitwise OR can be used. This operation is exact—it introduces no additional false positives compared to constructing a new Bloom filter from the union.

To check if a Bloom filter contains any element, we verify whether each partition contains at least one bit set to 1. If so, the filter is not empty.

The probability of false overlap during intersection is more complex to analyze. We refer interested readers to [11], which addresses this topic in detail. According to this source, for partitioned Bloom filters representing sets S_1 and S_2, the false overlap probability is:

$$\left(1 - \left(1 - \frac{k}{m}\right)^{|S_1||S_2|}\right)^k.$$

This probability can be quite high. For instance, intersecting two sets of size 16 yields the same false-positive rate as testing membership in a set of size 256. One saving grace, however, is that partitioned Bloom filters perform better than unpartitioned ones in hardware scenarios.

Bibliography

[1] *Advanced Synchronization Facility: Proposed Architectural Specification*. Revision 2.1. Advanced Micro Devices, Inc. Mar. 2009 (cit. on p. 10).

[2] Arvind. "Bluespec: A language for hardware design, simulation, synthesis and verification Invited Talk". *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. MEMOCODE '03. USA: IEEE Computer Society, 2003, p. 249. ISBN: 0769519237 (cit. on p. 11).

[3] B. H. Bloom. "Space/time trade-offs in hash coding with allowable errors". *Communications of the ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692 (cit. on p. 42).

[4] Z. Chen, A. Hassan, M. J. Kishi, J. Nelson, and R. Palmieri. "HaTS: Hardware-Assisted Transaction Scheduler". *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Vol. 153. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 10:1–10:16. ISBN: 978-3-95977-133-7. DOI: 10.4230/LIPIcs.OPODIS.2019.10 (cit. on p. 11).

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB". *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152 (cit. on pp. 11, 25).

[6] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. "A Reliable Randomized Algorithm for the Closest-Pair Problem". *Journal of Algorithms* 25.1 (1997), pp. 19–51. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1997.0873 (cit. on p. 32).

[7] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. "FaRM: fast remote memory". *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 401–414. ISBN: 9781931971096. DOI: 10.5555/2616448.2616486 (cit. on p. 33).

[8] K. Gopinathan and I. Sergey. "Certifying Certainty and Uncertainty in Approximate Membership Query Structures". *Computer Aided Verification*. Ed. by S. K. Lahiri and C. Wang. Cham: Springer International Publishing, 2020, pp. 279–303. ISBN: 978-3-030-53291-8. DOI: 10.1007/978-3-030-53291-8_16 (cit. on p. 42).

[9] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, Second Edition*. Springer Cham, June 2010. ISBN: 978-3-031-00600-5. DOI: 10.1007/978-3-031-01728-5 (cit. on p. 8).

[10] M. Herlihy and J. E. B. Moss. "Transactional memory: architectural support for lock-free data structures". *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, California, USA: Association for Computing Machinery, 1993, pp. 289–300. ISBN: 0818638109. DOI: 10.1145/165123.165164 (cit. on p. 10).

[11] M. C. Jeffrey and J. G. Steffan. "Understanding Bloom filter intersection for lazy address-set disambiguation". *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 345–354. ISBN: 9781450307437. DOI: 10.1145/1989493.1989551 (cit. on p. 45).

[12] H. T. Kung and J. T. Robinson. "On optimistic methods for concurrency control". *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567 (cit. on p. 10).

[13] M. Larabel. "Intel To Disable TSX By Default On More CPUs With New Microcode". *Phoronix* (June 28, 2021). URL: https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode (cit. on p. 10).

[14] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl. "There's Plenty of Room at the Top: What Will Drive Computer Performance after Moore's law?" *Science* 368.6495 (2020). ISSN: 0036-8075. DOI: 10.1126/science.aam9744 (cit. on p. 8).

[15] Z. Li, L. Liu, Y. Deng, J. Wang, Z. Liu, S. Yin, and S. Wei. "FPGA-Accelerated Optimistic Concurrency Control for Transactional Memory". *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 911–923. ISBN: 9781450369381. DOI: 10.1145/3352460.3358270 (cit. on p. 33).

[16] D. A. Menascé and T. Nakanishi. "Optimistic versus pessimistic concurrency control mechanisms in database management systems". *Information Systems* 7.1 (1982), pp. 13–27. ISSN: 0306-4379. DOI: 10.1016/0306-4379(82)90003-5 (cit. on p. 10).

[17] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. "STAMP: Stanford Transactional Applications for Multi-Processing". *2008 IEEE International Symposium on Workload Characterization*. 2008, pp. 35–46. DOI: 10.1109/IISWC.2008.4636089 (cit. on p. 11).

[18] D. Nguyen and K. Pingali. "What Scalable Programs Need from Transactional Memory". *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 105–118. ISBN: 9781450344654. DOI: 10.1145/3037697.3037750 (cit. on p. 11).

[19] Á. R. Perez-Lopez. "Puppetmaster: a certified hardware architecture for task parallelism". MA thesis. Cambridge, MA: Massachusetts Institute of Technology, Sept. 2021. DOI: 1721.1/140136 (cit. on pp. 9, 38).

[20] N. Shavit and D. Touitou. "Software transactional memory". *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada: Association for Computing Machinery, 1995, pp. 204–213. ISBN: 0897917103. DOI: 10.1145/224964.224987 (cit. on p. 10).

[21] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. "Calvin: fast distributed transactions for partitioned database systems". *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 1–12. ISBN: 9781450312479. DOI: 10.1145/2213836.2213838 (cit. on p. 33).

[22] *TPC Benchmark C Standard Specification*. Revision 5.11. Transaction Processing Performance Council. Feb. 2010. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf (cit. on p. 11).

[23] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores". *Proc. VLDB Endow.* 8.3 (Nov. 2014), pp. 209–220. ISSN: 2150-8097. DOI: 10.14778/2735508.2735511 (cit. on pp. 10, 11, 25, 29).