Puppetmaster: a certified hardware architecture for task parallelism

by

Áron Ricardo Perez-Lopez

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

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 13, 2021

Certified by . . . .
Adam Chlipala
Associate Professor of Computer Science
Thesis Supervisor

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

Puppetmaster: a certified hardware architecture for task parallelism

by

Áron Ricardo Perez-Lopez

Submitted to the Department of Electrical Engineering and Computer Science
on August 13, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents *Puppetmaster*, a hardware accelerator for transactional workloads. Existing software and hardware frameworks for transactional memory and online transaction processing are not able to scale to hundreds or thousands of cores unless the rate of conflicts between transactions is very low. *Puppetmaster* aims to improve upon the scalability of concurrency control by requiring transactions to declare their read and write sets in advance and uses this information to only run transactions concurrently when they are known not to conflict. In this thesis, I present and evaluate the design of *Puppetmaster* in a high-level model, in cycle-accurate simulations, and on real reconfigurable hardware.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

// Acknowledgments

I would like to thank Adam Chlipala, my thesis advisor, for trusting me, an undergrad with no knowledge of formal methods or hardware design, with carrying out a project like this and for supporting me throughout my journey. I would also like to thank Thomas Bourgeat for answering all my exciting and less exciting questions and for spending hours with me debugging mistakes in my code. I also thank the other members of the Programming Languages and Verification group for providing a research community throughout the pandemic and for helping me out with important questions about grad school and grad student life.

I thank all my teachers at MIT and beyond for being dedicated and caring, and I thank the 6.009 instructors of the past six semesters — Adam, my advisor, Srini Devadas, and Adam Hartz, among others — and Nora Watson at Cambridge Rindge and Latin School for trusting me with teaching others and helping me improve. I also thank the EECS department for providing TA funding and the staff in the EECS Undergraduate Office for helping all of us graduate and for being so awesome.

I am also grateful to my academic mentors throughout the years: Péter Csermely, who brought me into the world of research as a high school student and provided more support than I could have ever dreamed of; and Brad Pentelute, Chris Kaiser, Jason Miller, Saman Amarasinghe, and T. L. Taylor, my academic advisors at MIT, who kept me on track academically, allowed me to test my limits, and gave me life advice when I needed it the most.

I am eternally grateful to the communities at Number Six and *The Tech*, my adopted families, for providing food and study spaces, and to them and all my other friends at MIT for the many memories, friendships, and their moral and academic support. Last but not least, I also want to thank my family, my parents, my sister, and my partner for their love and unwavering support.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **15**

2 Background **17**

2.1 Detecting Conflicts in Concurrent Programs 17

2.2 Synthetic Transactional Workloads 19

2.3 Online Transaction Processing 21

2.4 Previous Work 23

3 Transaction Scheduling in Hardware **25**

3.1 Fast Comparisons via Object Renaming 26

3.2 Scheduling via Tournaments 29

4 Evaluation **31**

4.1 Experimental Setup 31

4.1.1 Input Generation 31

4.1.2 High-Level Model 32

4.1.3 Hardware Implementation 33

4.2 Results 34

4.2.1 Key-Value Store and Messaging Server 34

4.2.2 OLTP Benchmarks 36

4.3 Discussion 37

5 Proof of Correctness **39**

6 Conclusions and Future Work **43**

List of Figures

2-1 Data structures and tasks in the messaging-server workload 21

3-1 Datapaths for transactions through *Puppetmaster* 26

3-2 Renamer architecture, with objects currently being renamed underlined and requests/responses sent bolded 27

3-3 Tournament scheduling with paths for selected transactions bolded . 30

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

2-1 Transactions in the key-value-store workload. 20
2-2 Transactions in the messaging-server workload. 20

THIS PAGE INTENTIONALLY LEFT BLANK

List of Algorithms

2.1 Transfer money between two bank accounts. 17

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Moore's Law, first formulated by Gordon Moore in 1965, predicted that the number of transistors per processor would double about every two years. This held until the early 2000s and enabled the processing power of chips to grow accordingly [24]. After this growth slowed down, attention turned to parallel and distributed computing for further speedups.

The primary challenge of shared-memory parallel computing is the interference of threads of execution with each other. When multiple threads run simultaneously, their operations can interleave in unpredictable orders and yield incorrect results, as detailed in Section 2.1. It is possible to mitigate this problem by grouping operations (reads and writes to memory, potentially with some computation in between) into larger units that are executed atomically. These units are known as transactions, and an architecture with support for atomic execution of transactions is known as transactional memory [18].

Transactional memory (TM) implementations exist in software for many programming languages [8, 17, 19, 28, 45] but incur a significant overhead due to the bookkeeping involved in operating on transactions. This overhead can be reduced by implementing TM in hardware, but commercial implementations have been slow to materialize. AMD proposed a set of ISA extensions in 2009 that added support for TM, but this has not been implemented in any released processors [2]. Intel implemented TM under the name Transactional Synchronization Extensions (TSX) in

Haswell processors in 2013 and in all later generations up to Whiskey Lake in 2018, but these were all disabled due to bugs in the implementations, and later Intel CPUs don't support this feature at all [14, 22, 30, 42]. ARM added TM features to their instruction set in 2019, with no processors implementing it yet [26]. IBM's Blue Gene/Q supercomputers, released in 2012, included support for transactional memory [15], as did the POWER8 processor implementing the eponymous instruction set, released in 2014 [23].

Puppetmaster is a hardware accelerator implementing TM semantics. It is able to assist processing cores, the **puppets** (either forming part of the CPU or custom hardware), in executing a potentially infinite stream of unordered transactions with read and write sets declared in advance. *Puppetmaster* is able to extract high parallelism, scaling to hundreds of cores, from suitable workloads where individual transactions are as short as a few hundred CPU cycles (measured on a commodity processor).

Previous work has focused either on ordered parallelism, where transactions must be executed in a strict order to preserve correctness, or unordered parallelism with massive transactions that allow for relatively slow scheduling processes. *Puppetmaster* targets workloads without ordering constraints where transactions only perform a moderate amount of computation, such that the scheduling algorithm needs to be very efficient and run directly on purpose-built hardware. *Puppetmaster* could be employed in database servers or other systems which receive concurrent requests from millions of clients every second.

Chapter 2

Background

2.1 Detecting Conflicts in Concurrent Programs

A central topic of this thesis is the detection of conflicts between transactions or, more generally, threads of execution. But how do these conflicts arise, and why do we want to avoid them? This section describes some examples of such conflicts, also known as data races, and existing detection methods.

A classical example of a possible conflict is that of two threads trying to transfer money between accounts in a banking program. Let's assume that two threads execute Algorithm 2.1 concurrently, both transferring 50 units of money. One possible interleaving of operations is that Thread 1 executes line 2, but before it can reach line 3, Thread 2 executes lines 2–3 (finally, Thread 1 also reaches line 3). In this case, account_1 will end up having only 50 less money instead of 100 less, as intended! This is because the decrease operation on lines 2-3 is not atomic (even if written on a single line), i.e. another concurrent thread can execute operations in between.

```
1: function TRANSFER(account₁, account₂, amount)
2:     balance₁ ← account₁.GETBALANCE()
3:     account₁.SETBALANCE(balance₁ − amount)
4:     balance₂ ← account₂.GETBALANCE()
5:     acc₂.SETBALANCE(balance₂ + amount)
6: end function
```

Algorithm 2.1: Transfer money between two bank accounts.

There are multiple ways to solve these kinds of issues. One of them is using locks to "protect" shared resources. Only one thread is allowed to hold a given lock; all other threads trying to acquire that same lock are forced to wait or try again later. In this example, a lock could be acquired before manipulating any of the accounts — to prevent another thread from executing instructions in between the getBalance and setBalance operations — and released after the thread is done with that account. However, using a single lock to protect all accounts dramatically reduces the amount of parallelism that can be achieved. This issue can be solved by using separate locks to protect every account — this is called fine-grained locking — but managing a large amount of locks can consume additional resources. Operations to acquire and release locks also add overhead to the system, even if no other threads are running. An alternative to locks is the use of atomic operations. However, such operations require architectural support. In addition, lock-free data structures using atomics are notoriously hard to implement correctly.

Transactional memory (TM), and transactions in general, aim to solve these problems by introducing a different programming paradigm. Instead of writing traditional, serial programs that can be run on multiple threads, programmers can describe the tasks that need to be performed as transactions that are guaranteed to not interfere with each other. Here, "interference" means that one transaction attempts to read or modify an object that has already been modified by another transaction that is still running. An object is a region of arbitrary size in memory (the sizes supported depend on the implementation). The objects read and modified by a transaction comprise the transaction's read and write sets, respectively. If an object is both read and modified, it is only considered part of the write set; therefore, an object will never occur in both the read and write sets of a transaction.

For example, Algorithm 2.1 could be rewritten as two transactions, one that decreases the balance of account_1 and another one that increases the balance of account_2. If multiple transactions tried to modify the balance of the same account concurrently, only one of them would be allowed to proceed. The enforcement of such guarantees is the responsibility of the runtime system, either at the software or the hardware

level. While most modern languages have some level of support for software transactions, these implementations are only able to run long-running, computation-intensive transactions efficiently. For shorter transactions, they need hardware support, which is what *Puppetmaster* provides.

Typical hardware TM (HTM) implementations use optimistic concurrency control (OCC). This means that they let transactions start running as if they were regular code, but they track the read and write sets of each transaction. If a transaction performs a memory operation on the same object as another running transaction, and at least one of the two operations is a store/write, one of the two transactions is aborted. The main advantage of this approach is that nonconflicting transactions can proceed as if they were running "regular" code. However, when transactions perform writes and then are aborted, those modifications need to be rolled back. Alternatively, writes performed by transactions can be held back from being propagated to memory until the transactions successfully complete, but in that case there needs to be a separate commit phase that updates the memory contents. In addition, aborted transactions need to be restarted, so OCC works best for low-contention workloads, where there are relatively few conflicts.

This is the area where *Puppetmaster* aims to improve upon these HTM implementations. If transactions' read and write sets are computed in advance instead of being detected dynamically, it becomes possible to only run those transactions concurrently that are known to have no conflicts with each other. If this advance computation of conflicts can be done fast enough, higher throughput can be achieved than with OCC in high-contention workloads.

2.2 Synthetic Transactional Workloads

The in-advance conflict-detection approach described in Section 2.1 comes with a caveat: the read and write sets of transactions must be computed before they are allowed to execute. While such precomputation is not always possible, at least without significant changes to the code of the transactions themselves, we can find common

Type	Reads	Writes	Weight	Time (ns)
GET	1	0	1.00	75
SET	0	1	1.00	75
TRANSFER	0	2	0.50	300

Table 2-1: Transactions in the key-value-store workload.

Type	Reads	Writes	Weight	Time (ns)
FETCH	5	1	1.00	550
POST	0	2	0.10	700

Table 2-2: Transactions in the messaging-server workload.

tasks where this is the case. Two example workloads used throughout this thesis are a key-value store (simple database) and a messaging server (for a channel-based multiuser platform). The key-value store has smaller transactions than the messaging server, so given similar distributions of object "popularity," we can expect the former to have lower contention than the latter.

As shown in Table 2-1, there are three types of transactions in the key-value-store workload. GET reads and returns the value of a single object, SET writes a single object, and TRANSFER decreases the value of one object by a certain amount, while increasing the value of another object by the same amount (it has no read objects, because those are already in the write set, which allows reading the objects too). The frequency of transfer operations was set to half of the others, since presumably it is less frequently used.

In the messaging-server workload, each user follows a fixed number of channels (or threads) from which it can retrieve messages. Users can also post messages to any channel. These operations correspond to the two tasks shown in Table 2-2. Figure 2-1 shows how the two transactions interact with the data. However, since neither of the transactions knows in advance all objects it will access, they need to be implemented as two transactions each in practice. The PREPAREFETCH and PREPAREPOST transactions read the values necessary to determine which other objects will be accessed. The EXECUTEFETCH and EXECTEPOST transactions check that the retrieved values haven't changed and execute the actual operation. If the values were changed, they relaunch the "prepare phase" of the corresponding type.

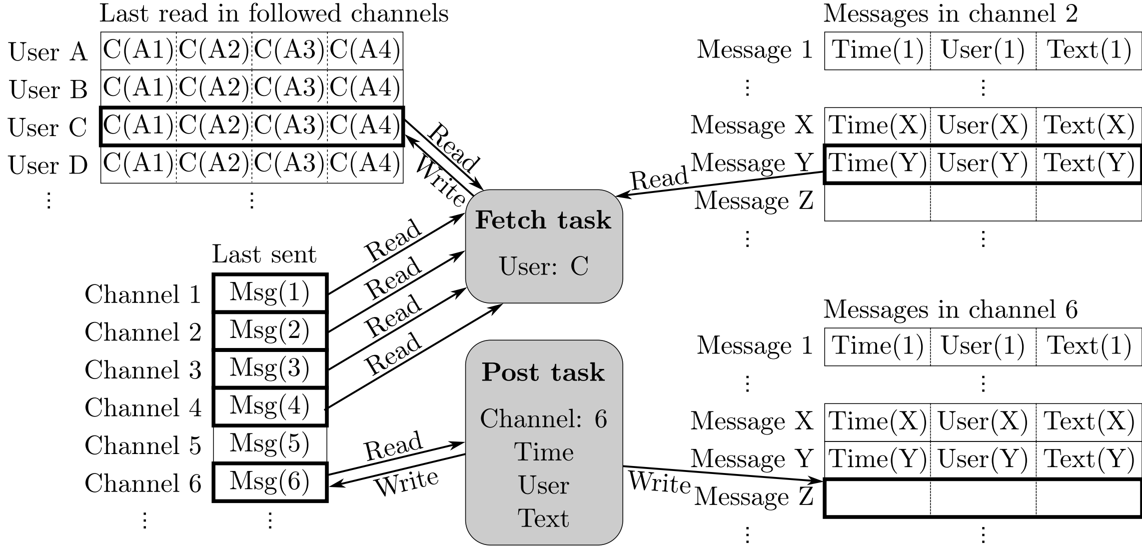

Figure 2-1: Data structures and tasks in the messaging-server workload

In order to measure the execution times of transactions in both workloads, I implemented them in Legion [31, 33]. Legion is a software-based transactional framework with semantics similar to *Puppetmaster* (read and write sets declared in advance) described in Section 2.4 in more detail. Since the original Legion implementation incurs significant overhead to ensure correctness when executing on multiple threads, I created a single-threaded implementation that does no concurrency control [32]. I measured the average running time of each type of transaction on a commodity processor (2.6 GHz 9th-generation Intel i7). The results are shown in Tables 2-1 and 2-2. The running time for the Fetch transactions was determined by combining the running times of the PrepareFetch and ExecuteFetch transactions (similarly for Post). When performing measurements in simulations of *Puppetmaster*, the Fetch and Post transactions were treated as single transactions for simplicity.

2.3 Online Transaction Processing

Online transaction processing (OLTP) systems need to be able to handle many short transactions concurrently. Typical examples are databases handling online orders and text messages, like the workloads in Section 2.2. Modern, commercial multi-

threaded databases can spend a third or more of the execution time of transactions on concurrency control (latching and locking) [16]. This can be even higher for logless or memory-resident databases that eschew logging and buffer management, respectively [16] (or both). In addition, currently used concurrency-control schemes are not able to scale to hundreds or thousands of processing cores, as demonstrated both in simulations [44] and on real multisocket hardware [3].

Although it has been demonstrated that certain well-implemented two-phase locking schemes are able to scale up to several hundred cores [4], a solution for high-contention workloads remains elusive. *Puppetmaster* aims to tackle this issue. Unlike many other concurrency-control schemes, *Puppetmaster* needs to know the read and write sets of a transaction before executing it. This requirement makes it impractical for some workloads, but such a strategy can yield very high throughput at the expense of somewhat higher latency. This was demonstrated in a study of deterministic databases which also require all objects affected by a transaction to be known in advance [38].

The two most commonly used benchmarks to measure OLTP database performance are TPC-C [41], published by the Transaction Processing Performance Council, and YCSB [9], the Yahoo! Cloud Serving Benchmark. TPC-C models an order-fulfillment system with multiple warehouses, customers, and items. It has "New Order" transactions — the only type found in all implementations of the benchmark, thus typically used for direct comparison — along with "Payment," "Order Status," "Delivery," and "Stock Level" transactions. YCSB has "Insert," "Update," "Read," and "Scan" transactions. The DBx1000 database engine implements the "New Order" and "Payment" transactions for TPC-C and the "Update," "Read," and "Scan" transactions for YCSB, as well as a suite of different concurrency-control algorithms. I extended DBx1000 to support offloading the concurrency control to hardware [36] and used this extended version to evaluate the performance of *Puppetmaster* relative to several software-based concurrency-control mechanisms.

2.4 Previous Work

Swarm [20] is an architecture that exploits ordered parallelism. It employs speculative execution, as the objects touched by each transaction (task) are not known in advance but are discovered as they are accessed. Transactions can also spawn other transactions. These "child transactions" are executed speculatively but are aborted if their parent transactions are aborted. This means that transactions need to keep track of their children, so there is a limit on the number of child transactions that can be spawned. Transactions are committed based on each transaction's global timestamp via infrequent communication with an arbiter unit.

Chronos [1] is an architecture that also exploits ordered parallelism speculatively. Chronos transactions can spawn new transactions; however, a transaction can only have a single read-write object (memory address) associated with it, known in advance. Having one object permits assigning transactions to processing units based on which objects they access. The authors demonstrate that this approach is equivalent to having transactions with unbounded read and write sets, since such transactions can be broken up into many single-object operations with subsequent timestamps.

Legion [5] is a parallel-programming framework with a hierarchical system of memory regions. Each transaction, starting with the top-level one, can spawn child transactions that can access a subset of the regions assigned to its parent, with equivalent or weaker permissions. Regions accessed by transactions need to be declared in advance. The software-based scheduling algorithm of Legion is fully distributed — each processing unit determines the destinations of the transactions spawned there — and takes into account heterogeneous systems with varying memory bandwidth and processor speed.

StarSs [37] is a parallel-programming model that allows declaring transactions that depend on each other. The transactions are functions with parameters marked as inputs and/or outputs, which allows the framework to dynamically discover dependencies between them and execute independent transactions in parallel. StarSs has been integrated into OpenMP [10], a standard framework for parallel programming,

under the name OmpSs [13]. StarSs/OmpSs are supported by multiple hardware accelerators [11, 12, 27, 29, 40, 43].

ROCoCoTM [25] offloads the validation phase of an optimistic concurrency-control algorithm to FPGAs. In ROCoCoTM, transactions are executed on the CPU, and their read and write sets are forwarded to the FPGA. The FPGA performs validation of transactions by implementing a reachability-based conflict-detection algorithm (ROCoCo). The read and write sets are represented in the form of Bloom filters (approximate bit vectors). Conflict detection for read-only transactions is performed on the CPU itself while they are running, and these transactions are committed immediately if not aborted.

Chapter 3

Transaction Scheduling in Hardware

Puppetmaster is a hardware-based transaction scheduler for unordered transactions. It sits between the processing units and the rest of the system. The processing units (puppets) can be general-purpose CPU cores or custom hardware capable of executing the kinds of transactions used by the workload. The input to the system is a stream of transactions from an external source (e.g. network). For each transaction, *Puppetmaster* receives the following information:

- **Transaction ID**: a unique identifier for the transaction.
- **Read set**: the set of objects read by the transaction.
- **Write set**: the set of objects written by the transaction.

As output, *Puppetmaster* produces IDs of transactions that are ready to start executing. These transactions are said to be scheduled. *Puppetmaster* guarantees that scheduled transactions don't conflict with any of the currently running transactions or each other. Conflict between two transactions is defined as an overlap between the write set of one transaction and the read or write set of the other. *Puppetmaster* can also signal that a transaction failed when some hardware resource (e.g. queue) associated with conflict resolution has been exhausted. In this case, it is the client's responsibility to restart the transaction. We expect that, usually, such transactions will be restarted automatically by the software sending transactions to *Puppetmaster*.

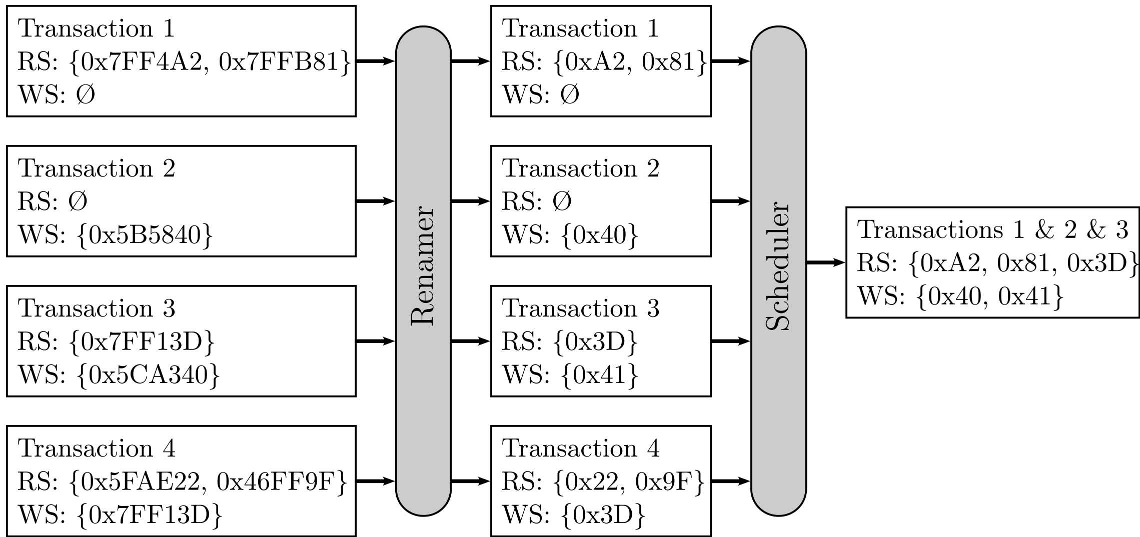

Figure 3-1: Datapaths for transactions through *Puppetmaster*

Scheduled transactions can be executed on any puppet that becomes available; this permits smart placement policies that minimize data movement between caches (see Chapter 6). The current implementation naïvely executes transactions on the first available puppets. The path of transactions through the system is summarized in Fig. 3-1.

Puppetmaster is aimed at workloads with transactions that take a few hundred to a few thousand clock cycles (0.1–1 microseconds) on a commodity processor. Given that reconfigurable hardware (FPGAs) available for implementation of the system has clock speeds at least an order of magnitude slower than commodity processors, there are only tens of cycles available in total for scheduling one transaction for each puppet. In order to achieve this speed, it is necessary both to be able to determine conflicts between any two transactions in at most tens of cycles, explored in Section 3.1, and to be able to schedule a number of transactions comparable to the number of puppets in the system, explored in Section 3.2.

3.1 Fast Comparisons via Object Renaming

A crucial step in the scheduling process is the detection of conflicts, which entails comparing the write sets of the two transactions with each other and the read set

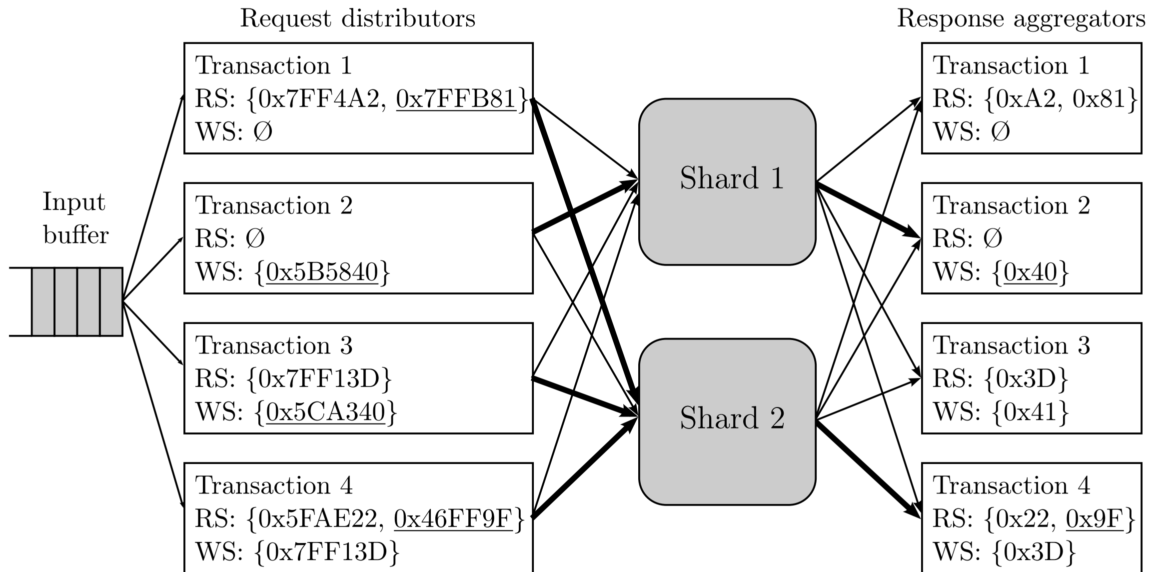

Figure 3-2: Renamer architecture, with objects currently being renamed underlined and requests/responses sent bolded

of one transaction with the write set of the other (in both directions). There is a conflict when any of the three set pairs have nonempty intersection. Of course, the three comparisons can be done in parallel, but parallelizing the intersection operation itself is more challenging. Elements can be compared pairwise if the sets are sorted. Implementing sorting efficiently in hardware is very complex and still quite slow, but encoding sets as bit vectors works just as well. In these vectors, the n-th bit is set if the value n is in the set. This allows for very fast set intersection and union operations, which just correspond to bitwise AND and OR, respectively, and operate on all elements of the set (all bits) in parallel.

However, operating on large bit vectors is still quite expensive in terms of area and latency (critical path length). At the same time, it is wasteful to use bit vectors that allow for storing memory addresses directly — for a 32-bit address space, this would result in vectors that are over 4 billion bits long! At any given time, assuming a small, finite bound on the sizes of read and write sets, there are many fewer addresses being used in the system. Therefore, we can save plenty of gates by using much smaller bit vectors and keeping track of which internal "index" or name corresponds to which real address.

To this end, *Puppetmaster* contains a renaming subunit, which maps memory addresses to object names. At a high level, the renaming unit takes each incoming transaction and performs the following steps for each object in the transaction's read and write sets:

1. Look up memory address in internal mapping table.
2. If the address already has an internal name associated with it, return that name.
3. Otherwise, create a new name candidate for the address, and look it up in the mapping table.
4. If there is no existing memory address associated with this name, return it, otherwise repeat the previous step.
5. If there are no more possible name candidates, signal that the transaction has failed.

Name candidates can be generated using a hashing scheme. A simple but effective scheme, which I implemented, is taking the lowest-order k bits of the address for the first candidate — where k depends on the size of the internal names, which is a tunable parameter of the system — and adding one each time to generate new candidates. (In practice, since memory addresses are usually word-aligned, the first 2–3 bits are not included in the generated hash value to reduce collisions.)

In order to allow for renaming objects in multiple transactions in parallel, the renaming subunit includes multiple renaming modules, as shown in Fig. 3-2. To reduce pressure on the RAM block containing the renaming table (mappings between memory addresses and internal names), it is split into several shards. Each of the n shards is responsible for $1/n$ of the address space, and objects are assigned to them using an additional hash function, which currently selects the lowest-order bits of the address above the ones used by the hash functions inside the shards, but more sophisticated versions are possible. The parallel renaming units are connected to all shards via a crossbar.

3.2 Scheduling via Tournaments

As described in the introduction to this chapter, *Puppetmaster* must schedule about the same number of transactions as there are puppets. This throughput can be achieved through the use of **tournaments**, a reduction-tree-like structure that computes conflicts among pairs of transactions, shown in Fig. 3-3.

One tournament consists of multiple **rounds**. In each round, transactions are compared pairwise. If the two transactions in a pair are determined to be compatible, their read and write sets are merged, and they behave as a single, larger transaction in further rounds. (Such transactions keep track which original transactions they were created from.) Otherwise, only one of the transactions (the first one according to the initial order in the current implementation) progresses to the next round. If there are an odd number of transactions, the pairless one proceeds to the next round automatically.

The compatibility of two transactions is determined by computing the intersections of the read set of the first transaction with the write set of the second, the write set of the first with the read set of the second, and the write set of the first with the write set of the second. If these are all empty, the transactions are compatible. The intersections are computed using bitwise AND operations on the bit-vector representations of the sets.

After $\lceil \log n \rceil$ rounds, where n is the initial number of transactions, there will only be a single transaction left and the tournament ends. If this transaction was created from the merger of multiple original transactions, those are all compatible with each other by construction. In order to avoid conflicts with running transactions, those are merged into a single transaction before the start of the tournament and are used as the highest-priority transaction. (The exact location of the highest-priority transaction depends on the implementation. With the conflict-resolution strategy described above, this is the first slot.) The set of transactions contained in the last remaining (merged) transaction, less the set of running transactions, is returned by *Puppetmaster*.

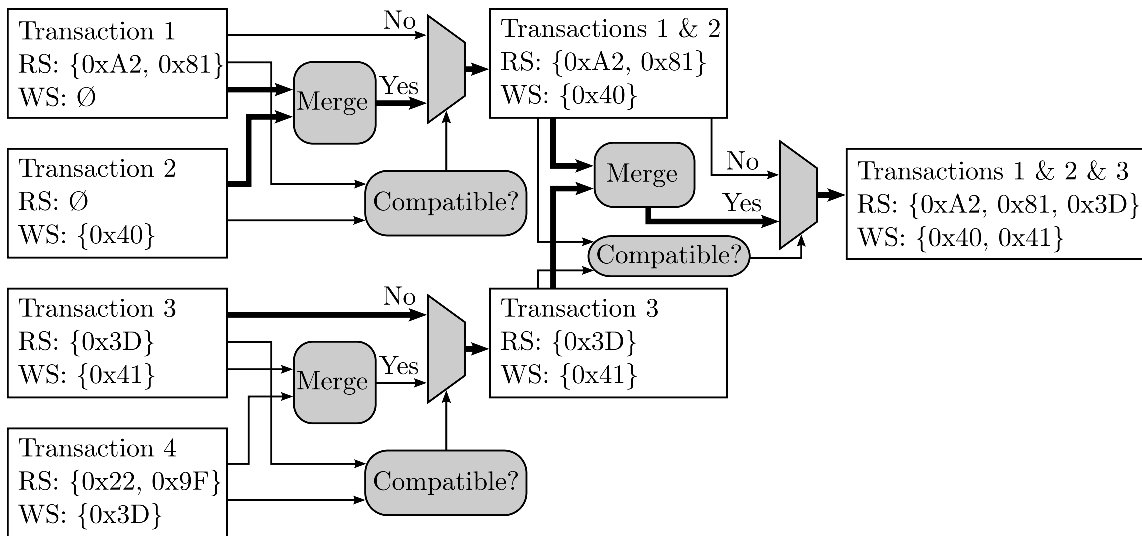

Figure 3-3: Tournament scheduling with paths for selected transactions bolded

Chapter 4

Evaluation

I evaluated the performance of *Puppetmaster* on two groups of benchmarks: the key-value-store and messaging-server workloads presented in Section 2.2 and the industry-standard OLTP benchmarks presented in Section 2.3, TPC-C and YCSB. The architecture described in Chapter 3 was evaluated at three different levels of abstraction: in a high-level model of the system written in Python, in cycle-accurate simulations of synthesizable Verilog, and on a commercial reconfigurable hardware device or FPGA.

4.1 Experimental Setup

4.1.1 Input Generation

Input transactions for the key-value-store and the messaging-server workloads were generated using a custom Python script, part of the high-level simulator [35]. The script can generate transactions for either of the workloads in Section 2.2, but configuring custom ones is also possible by specifying the types of transactions to be generated, including the sizes of their read and write sets and their relative frequency. The script allows specifying the memory size, i.e. the number of objects that can be referenced by the transactions. This influences the frequency of conflicts between transactions: the smaller the memory, the fewer total addresses there are available and the more likely it is for two transactions to access the same address.

The workloads targeted by *Puppetmaster* realistically have between $2^{16} = 65,536$ and $2^{24} = 16,777,216$ objects.

The script also allows specifying the probability distribution of picking the objects (their "popularity") by means of the parameter for a Zipfian distribution, where a parameter of 0 results in the uniform distribution and a parameter of 1 results in a distribution where the rank of an object is inversely correlated with its frequency. The larger this parameter is, the more frequent conflicts between transactions become.

Input transactions for TPC-C and YCSB were generated using DBx1000 [36].

4.1.2 High-Level Model

The high-level Python model of *Puppetmaster* approximates the behavior of the physical system using estimates of the running time of the scheduling algorithm and the execution time of transactions. The running time of the scheduling algorithm is calculated by multiplying the number of steps (corresponding to clock cycles) taken by the scheduler by the clock period, a parameter to the model. The values shown in Tables 2-1 and 2-2 were used as execution times. The OLTP benchmarks were not modeled.

The model implements the tournament-based algorithm described in Section 3.2. For comparison, it also features an algorithm that returns the largest subset of compatible transactions. However, this algorithm has exponential time complexity in the number of transactions considered, therefore, for most measurements a linear-time greedy algorithm was used instead, which performs nearly as well as the maximal algorithm. The greedy scheduling algorithm considers transactions one at a time and selects them if they are compatible with already selected transactions. This algorithm is assumed to complete scheduling in a single step (clock cycle), while the tournament-scheduling algorithm takes a variable amount of steps, just like in real hardware, depending on the number of transactions considered (the "pool size") and the number of comparator units available, which are both configurable parameters. The maximum number of transactions waiting to be executed or the "queue size" can also be adjusted.

The pool size influences the area of the hardware substantially in addition to the scheduling time. Values of up to 128 were evaluated, which is about the upper bound of what can be implemented in practice. Increasing the queue size allows the scheduler to be run repeatedly while waiting for the puppets to finish executing, and in real hardware it helps hide communication latency between the CPU and the custom logic. The queue can be implemented with a simple FIFO, so it can be much larger that the scheduling pool, but a queue size of 128 is enough for the configurations tested.

The model can represent read and write sets as unbounded hash tables that cannot be efficiently implemented in hardware, but it can also emulate more realistic fixed-size sets based on bit vectors. It can model approximate sets (similar to Bloom filters [6]) that have a bit set for each value based on a hash function (implemented as modulus) or a more sophisticated construction that uses a global renaming table, as described in Section 3.1. The size (number of bits) of these latter sets can be adjusted via a parameter; set sizes up to 2^{10} were tested, which is about the largest that can be reasonably fit on commercial FPGAs with all the necessary logic to manipulate them.

4.1.3 Hardware Implementation

The synthesizable implementation of *Puppetmaster* follows the design described in Chapter 3. I wrote it in Bluespec SystemVerilog [34], which compiles to plain Verilog and can be used to build cycle-accurate simulations using Verilator [39] as well as bitstreams for reconfigurable hardware (FPGA). The Connectal [21] toolkit was used to connect the hardware to programs (written in C++) running on the CPU. For the purposes of running the key-value-store and messaging-server workloads, puppets that simply wait a preset number of clock cycles (depending on transaction type) were added to the hardware design.

As described in Section 2.3, database engines generally spend at least a third of the execution time dealing with concurrency control, and even more under high contention. With a fast enough hardware-based scheduler, it is thus possible to reduce the time spent on each transaction significantly. I adapted the DBx1000 database

implementation to use an external concurrency-control mechanism and connected it to *Puppetmaster*. Transactions, instead of waiting for locks held by other threads then executing, are sent to *Puppetmaster*. Since it would have been challenging to implement puppets capable of executing database transactions directly in hardware, transactions that are ready to run are sent back to the main processor and executed there.

4.2 Results

For the key-value-store and messaging-server benchmarks, the parallelism was measured using the high-level simulator. In Verilator simulations of the hardware as well as runs on the actual FPGA, transaction throughput was measured. The parallelism is the total execution time of transactions, when run serially, divided by the actual running time of the system. It is equal to the average number of transactions running simultaneously. Parallelism was computed for steady state, so excluding the initial warm-up (when the internal queues are being filled) and final taper-off (when there are no more transactions left) phases, which approximates the parallelism for an infinite stream of transactions. The throughput in transactions per second (TPS) can be calculated by dividing the number of transactions by the running time or, alternatively, by dividing the parallelism by the average execution time of transactions. In Verilator simulations, the latency distribution of transactions was also measured.

4.2.1 Key-Value Store and Messaging Server

The maximum achievable parallelism in the key-value-store and messaging-server workloads can be measured in the high-level simulator using the greedy scheduling algorithm, unbounded sets, and the largest reasonable parameter for pool size (128). Under low contention (2^{24} addresses), this parallelism is over 2000 for both workloads when a "zero-period" clock is used, i.e. scheduled transactions are immediately available. More realistic clock periods are 8/16/32 ns. When using these, parallelism decreases to 1500/900/500, respectively, for the key-value store but stays over 2000

for the messaging server. With the same parameters, the tournament scheduler is able to extract parallelisms of 350/180/150 for the key-value store and 1300/650/400 for the messaging server.

Using 1024-bit fixed-size sets with a global renaming table yields similar results to those with the idealized set for the key-value store but decreases parallelism to 180/150/120 for the messaging server. Using approximate sets decreases the parallelism to 110/60/50 for the key-value store and 110/90/50 for the messaging server. Under higher contention, with an address space size of 2^{16}, the theoretical maximum parallelism (calculated using a zero-period clock) with the greedy algorithm decreases to 1150 for the messaging server, but the tournament scheduler is able to achieve similar performance as under lower contention.

When the object distribution is Zipfian with a parameter of 0.5 and the memory size is 2^{24}, the tournament scheduler performs just as well for both workloads as when the distribution is uniform. When the memory size is 2^{16}, parallelism decreases by 10-15%. For very skewed object distributions with a Zipf parameter of 1, the performance of the tournament scheduler degrades significantly, yielding parallelisms of less than 30 and 15 for the key-value-store and messaging-server workloads, respectively, with a memory size of 2^{24}, and less than 15 and 8 with a memory size of 2^{16}. However, for this distribution, even using unbounded sets or the greedy algorithm does not help.

Reducing the pool size limits the parallelism that can be achieved by the tournament scheduler. In most cases this means that the pool size needs to be at least half the desired parallelism. In cases of particularly low contention, the pool size can be a quarter or even less of the desired parallelism. Reducing the size of the set representations similarly limits the parallelism that can be achieved by limiting the maximum number of objects in flight (objects in all renamed transactions that haven't finished yet).

In Verilator simulation, *Puppetmaster* achieves a throughput of 3.3 million TPS for both workloads when using a clock period of 8 ns and uniform object distribution. This is much lower than the predicted throughput of around 2 billion TPS for these workloads due to the fact that the hardware spends much of the time waiting for the

transactions to be transferred from the software to the hardware. The median latency of transactions in both workloads is 140–150 cycles or 1.1–1.2 μs. 95% of transactions had latencies under 250 cycles or 2 μs. The key-value-store and messaging-server workloads were not tested on a physical FPGA, since all processing takes place in the custom hardware of *Puppetmaster*, so the results are expected to be the same as those from the Verilator simulation.

4.2.2 OLTP Benchmarks

The OLTP benchmarks TPC-C and YCSB were only evaluated with the synthesized hardware design, since the high-level model assumes that all processing takes place inside *Puppetmaster* and does not account for the additional latency between the FPGA and the CPU when executing transactions. Moreover, the transactions generated by DBx1000 are not compatible with the model.

TPC-C and YCSB achieve a throughput of 5.3 million TPS in the Verilator simulation assuming a clock period of 5 ns, corresponding to the FPGA is running at 200 MHz, which is the highest frequency at which the design has been successfully synthesized. However, this value was calculated in the time frame of the hardware, which is not accurate, because the Verilator simulation runs much slower than real hardware, so database transactions appear much faster.

To get accurate throughput measurements, the OLTP benchmarks were run on Amazon EC2 F1 instances with 2.3 GHz Intel Xeon E5-2686 v4 processors connected to Xilinx Virtex UltraScale+ VU9P FPGAs. However, the throughput was much lower than expected and much lower than with software-only concurrency control. Based on timing data from the FPGA, the renaming and scheduling time of transactions is very similar to those measured in the simulation, as expected, but additional delays are present before transactions are renamed.

4.3 Discussion

As can be seen from the results, the tournament-scheduler algorithm is able to scale to several hundred cores in the best case and more than a hundred cores even in less optimal cases. When the object distribution is very skewed, the algorithm's performance decreases significantly, but even more optimal algorithms can't do better. These results hold as long as the pool size is chosen to be at least half the target parallelism and renaming-based bounded sets are used with a size in accordance with the expected number of objects in flight.

The key-value-store workload, which has much smaller transactions, allows for smaller parallelism in the best case, but its performance degrades significantly less with increased object contention (smaller memory size and skewed object distribution) than that of the messaging-server workload.

The hardware implementation of *Puppetmaster* doesn't perform as well as in models. The causes of this need to be investigated in future work. The most likely culprits are delays introduced by limitations in the connection between the software and the hardware side.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Proof of Correctness

No system, however fast, is useful without also being correct. As mentioned in the introduction, Chapter 1, commercial hardware TM implementations have been plagued by correctness issues. Therefore, it is crucial to be able to confirm the correctness of the design of *Puppetmaster*. I verified the design in the Coq theorem prover.

The design is represented as a transition system with a state comprising five sets of transactions: Queued (added to the system, no action taken yet), Renamed, Scheduled, Running, and Finished. The state transitions in the system are the following:

1. **Add**: a new transaction is added to Queued.
2. A transaction is moved from Queued to Renamed.
3. A group of transactions of any size is selected from Renamed, and the compatible transactions get moved to Scheduled, as determined by the tournament-scheduling algorithm described in Section 3.2.
4. **Start**: a transaction in Scheduled is moved to Running.
5. **Finish**: a transaction in Running is moved to Finished.

Executions in this transition system generate traces where transitions 2 and 3, renaming and scheduling, are silent, but transitions 1, 4, and 5 correspond to the actions "Add," "Start," and "Finish." The actual process of renaming objects is not

currently modeled; transition 2 leaves the object names/addresses inside transactions intact.

The main theorem states that any trace generated by this transition system can also be generated by the specification of the system, starting from an initial state where all sets are empty. The specification is another transition system with a state comprising three sets: Queued, Running, and Finished. It has the following transitions and associated actions:

1. **Add**: a new transaction is added to Queued.
2. **Start**: a transaction in Queued that is compatible with all transactions already in Running is moved to Running.
3. **Finish**: a transaction in Running is moved to finished.

The main theorem can be proved with the help of an invariant of the implementation's transition system. The invariant is a property about the state that holds after every transition if it also held before the transition. For *Puppetmaster*, the invariant asserts that all transactions in Scheduled are compatible with each other, all transactions in Running are compatible with each other, and every transaction in Scheduled is compatible with all transactions in Running.

In addition to the invariant, the proofs also use a simulation relation. This relation establishes which implementation and specification states correspond to each other. For *Puppetmaster*, an implementation state corresponds to a specification state if the implementation state's Queued, Renamed, and Scheduled sets together contain the same transactions as the specification state's Queued set (and there are no duplicates), and the Running and Finished sets of the implementation state are equal to the Running and Finished sets of the specification state, respectively.

Using the implementation state invariant, we can prove a refinement lemma. This lemma states that for every implementation trace there is a corresponding specification trace where the initial and final states of the implementation and the initial and final states of the specification are related to each other by the simulation relation. The main theorem follows directly from this lemma.

The proof of the refinement lemma uses structural induction over the implementation traces. The cases for transitions 1, 2, 4, and 5 are simple to prove, but proving transition 3, the scheduling process, requires a couple of auxiliary lemmas. These state that the tournament-scheduling algorithm always returns the first input transaction as one of the outputs, that every round of scheduling halves the number of (possibly merged) transactions in the scheduler, and that the final, merged transaction was created from transactions that are compatible with each other.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Conclusions and Future Work

The work in this thesis demonstrates that *Puppetmaster*, a transaction scheduler implemented in hardware, can scale to hundreds of processing cores when all objects affected by transactions are known in advance.

Further work is necessary to confirm whether the predictions from models can be achieved in practice. It also remains to be seen whether larger instances of *Puppetmaster* that can only fit on FPGAs larger than the ones available now can achieve better performance. Optimizing the hardware design, including pipelining long combinational paths, could yield further speedups. Utilizing an FPGA closely integrated with the CPU, as presented by Morais et al. [29], could reduce latency significantly.

Possible improvements in the formal methodology used to verify the correctness of the design include a more realistic renaming procedure. The current model simply leaves object addresses as-is, and implementing the renaming table used in the hardware design would allow for higher confidence in the correctness of the design. Going even further, the hardware design could be implemented in Kôika [7] or a similar language where formal reasoning is possible directly over the compiled design.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Maleen Abeydeera and Daniel Sanchez. "Chronos: Efficient Speculative Parallelism for Accelerators". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1247–1262. ISBN: 9781450371025. DOI: 10.1145/3373376.3378454 (cit. on p. 23).

[2] *Advanced Synchronization Facility: Proposed Architectural Specification*. Revision 2.1. Advanced Micro Devices, Inc. Mar. 2009. URL: http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf (cit. on p. 15).

[3] Tiemo Bang et al. "The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware". In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. DaMoN '20. Portland, Oregon: Association for Computing Machinery, 2020. ISBN: 9781450380249. DOI: 10.1145/3399666.3399910 (cit. on p. 22).

[4] Claude Barthels et al. "Strong Consistency is Not Hard to Get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores". In: *Proc. VLDB Endow.* 12.13 (Sept. 2019), pp. 2325–2338. ISSN: 2150-8097. DOI: 10.14778/3358701.3358702. URL: https://doi.org/10.14778/3358701.3358702 (cit. on p. 22).

[5] Michael Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71 (cit. on p. 23).

[6] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692 (cit. on p. 33).

[7] Thomas Bourgeat et al. "The Essence of Bluespec: A Core Language for Rule-Based Hardware Design". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. ISBN: 9781450376136. DOI: 10.1145/3385412.3385965 (cit. on p. 43).

[8] *C/C++ Language Constructs for TM*. The GNU Compiler Collection. URL: https://gcc.gnu.org/onlinedocs/libitm/C_002fC_002b_002b-Language-Constructs-for-TM.html (visited on 08/2021) (cit. on p. 15).

[9] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *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 p. 22).

[10] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313 (cit. on p. 23).

[11] Tamer Dallou and Ben Juurlink. "Hardware-Based Task Dependency Resolution for the StarSs Programming Model". In: *2012 41st International Conference on Parallel Processing Workshops*. 2012, pp. 367–374. DOI: 10.1109/ICPPW.2012.53 (cit. on p. 24).

[12] Tamer Dallou et al. "Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 1129–1138. DOI: 10.1109/IPDPS.2015.79 (cit. on p. 24).

[13] Alejandro Duran et al. "OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures". In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193. DOI: 10.1142/S0129626411000151 (cit. on p. 24).

[14] Gareth Halfacree. "Intel Sticks Another Nail in the Coffin of TSX with Feature-Disabling Microcode Update". In: *The Register* (June 29, 2021). URL: https://www.theregister.com/2021/06/29/intel_tsx_disabled/ (cit. on p. 16).

[15] Ruud Haring et al. "The IBM Blue Gene/Q Compute Chip". In: *IEEE Micro* 32.2 (2012), pp. 48–60. DOI: 10.1109/MM.2011.108 (cit. on p. 16).

[16] Stavros Harizopoulos et al. "OLTP through the Looking Glass, and What We Found There". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 981–992. ISBN: 9781605581026. DOI: 10.1145/1376616.1376713 (cit. on p. 22).

[17] Tim Harris, Simon Marlow, and Simon Peyton Jones. "Composable Memory Transactions". In: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, Jan. 2005, pp. 48–60. ISBN: 1-59593-080-9. URL: https://www.microsoft.com/en-us/research/publication/composable-memory-transactions/ (cit. on p. 15).

[18] M. Herlihy, J. Eliot, and B. Moss. "Transactional Memory: Architectural Support For Lock-free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 1993, pp. 289–300. DOI: 10.1109/ISCA.1993.698569 (cit. on p. 15).

[19] Rich Hickey. "The Clojure programming language". In: *Proceedings of the 2008 symposium on Dynamic languages*. 2008, pp. 1–1 (cit. on p. 15).

[20] Mark C. Jeffrey et al. "A Scalable Architecture for Ordered Parallelism". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 228–241. ISBN: 9781450340342. DOI: 10.1145/2830772.2830777 (cit. on p. 23).

[21] Myron King, Jamey Hicks, and John Ankcorn. "Software-Driven Hardware Development". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 13–22. ISBN: 9781450333153. DOI: 10.1145/2684746.2689064 (cit. on p. 33).

[22] Michael Larabel. "Intel To Disable TSX By Default On More CPUs With New Microcode". In: *Phoronix* (June 28, 2021). URL: https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode (cit. on p. 16).

[23] H. Q. Le et al. "Transactional memory support in the IBM POWER8 processor". In: *IBM Journal of Research and Development* 59.1 (2015), 8:1–8:14. DOI: 10.1147/JRD.2014.2380199 (cit. on p. 16).

[24] Charles E. Leiserson et al. "There's Plenty of Room at the Top: What Will Drive Computer Performance after Moore's law?" In: *Science* 368.6495 (2020). ISSN: 0036-8075. DOI: 10.1126/science.aam9744. eprint: https://science.sciencemag.org/content/368/6495/eaam9744.full.pdf. URL: https://science.sciencemag.org/content/368/6495/eaam9744 (cit. on p. 15).

[25] Zhaoshi Li et al. "FPGA-Accelerated Optimistic Concurrency Control for Transactional Memory". In: *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. 24).

[26] Berenice Mann. "New Technologies for the Arm A-Profile Architecture". In: *Arm Community Blog* (Apr. 18, 2019). URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture (cit. on p. 16).

[27] Cor Meenderinck and Ben Juurlink. "A Case for Hardware Task Management Support for the StarSS Programming Model". In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010, pp. 347–354. DOI: 10.1109/DSD.2010.63 (cit. on p. 24).

[28] Remigius Meier and Thomas R. Gross. "Reflections on the Compatibility, Performance, and Scalability of Parallel Python". In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 91–103. ISBN: 9781450369961. DOI: 10.1145/3359619.3359747 (cit. on p. 15).

[29] Lucas Morais et al. "Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-Core Processor". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 861–872. ISBN: 9781450369381. DOI: 10.1145/3352460.3358271 (cit. on pp. 24, 43).

[30] Shaun Nichols. "True to Its Name, Intel CPU Flaw ZombieLoad Comes Shuffling Back with New Variant". In: *The Register* (Nov. 12, 2019). URL: https://www.theregister.com/2019/11/12/zombieload_cpu_attack/ (cit. on p. 16).

[31] Áron Ricardo Perez-Lopez. *CyanoKobalamyne/kvs-legion: First release*. Version v1.0. Sept. 2021. DOI: 10.5281/zenodo.5521498 (cit. on p. 21).

[32] Áron Ricardo Perez-Lopez. *CyanoKobalamyne/legion-serial: First release*. Version v1.0. Sept. 2021. DOI: 10.5281/zenodo.5521500 (cit. on p. 21).

[33] Áron Ricardo Perez-Lopez. *CyanoKobalamyne/msgserver-legion: First release*. Version v1.0. Sept. 2021. DOI: 10.5281/zenodo.5521496 (cit. on p. 21).

[34] Áron Ricardo Perez-Lopez. *CyanoKobalamyne/pmhw: First release*. Version v1.0. Sept. 2021. DOI: 10.5281/zenodo.5521488 (cit. on p. 33).

[35] Áron Ricardo Perez-Lopez. *CyanoKobalamyne/pmsim: First release*. Version v1.0. Sept. 2021. DOI: 10.5281/zenodo.5521492 (cit. on p. 31).

[36] Áron Ricardo Perez-Lopez et al. *CyanoKobalamyne/DBx1000: First release*. Version v1.0. Sept. 2021. DOI: 10.5281/zenodo.5521486 (cit. on pp. 22, 32).

[37] Judit Planas et al. "Hierarchical Task-Based Programming With StarSs". In: *The International Journal of High Performance Computing Applications* 23.3 (2009), pp. 284–299. DOI: 10.1177/1094342009106195 (cit. on p. 23).

[38] Kun Ren, Alexander Thomson, and Daniel J. Abadi. "An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems". In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 821–832. ISSN: 2150-8097. DOI: 10.14778/2732951.2732955 (cit. on p. 22).

[39] Wilson Snyder. *Verilator*. 2021. URL: https://www.veripool.org/verilator/ (cit. on p. 33).

[40] Xubin Tan et al. "General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 244–253. DOI: 10.1109/IPDPS.2017.48 (cit. on p. 24).

[41] *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. 22).

[42] Scott Wasson. "Errata Prompts Intel to Disable TSX in Haswell, Early Broadwell CPUs". In: *The Tech Report* (Aug. 12, 2014). URL: https://techreport.com/news/26911/errata-prompts-intel-to-disable-tsx-in-haswell-early-broadwell-cpus/ (cit. on p. 16).

[43] Fahimeh Yazdanpanah et al. “Picos: A hardware runtime architecture support for OmpSs”. In: *Future Generation Computer Systems* 53 (2015), pp. 130–139. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2014.12.010. URL: https://www.sciencedirect.com/science/article/pii/S0167739X14002702 (cit. on p. 24).

[44] Xiangyao Yu et al. “Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores”. In: *Proc. VLDB Endow.* 8.3 (Nov. 2014), pp. 209–220. ISSN: 2150-8097. DOI: 10.14778/2735508.2735511 (cit. on p. 22).

[45] Thomas Zimmermann. *picotm — Portable Integrated Customizable and Open Transaction Manager*. URL: http://picotm.org/ (cit. on p. 15).