MEng Individual Project

Imperial College London

Department of Computing

Deterministic concurrency control for transaction processing systems on FPGAs

Author:
William Woodacre

Supervisor:
Dr. Holger Pirk

Second Marker:
Dr. Alexandros Koliousis

June 17, 2019

Abstract

Transactions are the unit of change for a database system. With the rise of e-commerce and other *internet scale* services, the role of online transaction processing (OLTP) systems has become increasingly important. In parallel to this, the slowdown of single core performance improvements has lead to many data-centres utilising increasingly heterogeneous hardware, including the use of FPGAs. However whilst there has been research into the applications of FPGAs in acceleration various aspect of database systems such as query processing [36] or caching [3], there has not been any investigation into transaction processing; the heart of a database management system (DBMS).

To remedy this, this project will investigate transaction processing of OLTP workloads using an FPGA. This project implements both deterministic concurrency control as well as more traditional two phase locking (TPL) both on the CPU and FPGA as a basis of comparison.

While deterministic concurrency control is a promising new technology for multi-node databases [30], the finding of the project show that it's single node performance does not scale well to utilise the resources of a single server effectively. However when transaction processing is moved to an FPGA and the cost of synchronisation is large as in a multi-node system, we see that this method of concurrency control far outperforms TPL. This project also introduces a novel deterministic lock manager, suited to batch transaction processing on FPGAs. These improvements combined with the speed and parallelism possible on an FPGA produced better performance than seen from that of the purely CPU based implementations.

From our findings, the use of FPGAs for single node in memory transaction processing shows much promise. This project has the potential to form the basis of future investigations into the application of FPGAs in transaction processing.

Acknowledgements

I would like to thank the following people for their support through this project:

- **Holger Pirk** for his support, encouragement and invaluable guidance throughout the duration of this project.
- **Alexandros Koliousis** for his feedback and advice on the direction of the project.
- **Dom Margan** and **George Theodorakis** in the LSDS group for their assistance in helping me to set up and configure the FPGA hardware.
- **Intel** for the generous donation of the FPGA boards used during this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transactions are the unit of change for a database system. With the rise of e-commerce and other *internet scale* services, the role of transaction processing systems has become increasingly important. Even very small delays in these services can cause significant damage to these buisnesses. In tests done by Amazon.com, it was found that small delays of even 100 miliseconds could cause substantial drops in revenue [24]. The requirement for such systems has created a need for a new class of database systems known as OLTP or on-line transaction processing databases. OLTP systems typically process short-lived and repetitive transactions [19], but have to scale to handle a high throughput of these transactions at a reasonable latency at peak times.

The main challenge for OLTP systems is to maximise the throughput of transactions while keeping a serial order in order to maintain ACID properties (as defined in A.2). A key problem achieving this is trying to minimise the number of acquired locks for a transaction (to maintain high levels of concurrency) while simultaneously minimising communication between nodes.

The classical way that distributed databases have been architected to solve this problem was pioneered by System R* [25] in the 1980s. This popularised two-phase commit which is still widely used today. In two-phase commit, all of the participating nodes in a distributed transactions have to agree whether the transaction was successful. The problem with this methodology is that the cost of communication is large as this involves many network round trips between all participating machines which can often take longer than the actual transaction processing. This is a problem that has typically plagued the scalability of distributed database systems.

Because of these problems, there has been a significant movement to relax some of the ACID guarantees in order to yield better performance and scalability. Amazon's Dynamo [6] relaxes consistency guarantees, while Google's Bigtable [4] completely removes support for multi-row transactions. This can be well suited to some applications, however, a large amount of applications do not fit these models well. In many applications database transactions mirror physical transactions, e.g. money moving from one account to others or a warehouses stock being added to or taken from. If ACID properties are relaxed then serious issues can occur such as persons spending money they do not have or customers ordering items that do not exist. Moreover, relaxing ACID can often make it harder for developers to reason about their applications which can slow down development and cause bugs.

With the continued increase of single server performance and availability of main memory, the time taken to execute transactions is continually decreasing. In 2010, it was suggested in a paper [30] that deterministic concurrency control be revisited as with low transaction times the overhead of determinism is almost negligible whilst removing the problem of synchronisation overhead. This was then incorporated into the system Calvin [31] in 2012. The result was a system that out-preformed most popular concurrency control schemes in a number of key benchmarks [11].

As Moore's law nears its end [34], modern data-centres are increasingly more heterogeneous in order to keep up with the demands of applications. Microsoft's Azure data centres have a FPGAs on every server since late 2015 [8], as well as offering GPU services. There has also been a large interest in the database community to use more heterogeneous hardware to accelerate various aspects of database systems. Examples of this include query processing [36] and caching [3].

However much of this work has been focused on using additional hardware to accelerate current system, and not to design a new system around a more heterogeneous architecture. This begs the question, how would a database look if it were built around modern heterogeneous systems? This is the question this project intends to answer.

1.1 Objectives

Given transaction processing is at the core of a DBMS, we investigate how transaction processing can be done utilizing an FPGA. We experiment with different types of currency control to discover what method is optimal for transaction processing on FPGAs, including deterministic concurrency control.

The primary objective of this project is to demonstrate through this investigation that FPGAs can offer improvements in transaction processing over a purely CPU based implementation. Our secondary objective is to provide a throughout understanding of what works well for transaction processing on an FPGA, as well as the limitations.

The hypothesis of this project is that transaction processing on a single node can be accelerated significantly by a highly parallel design possible on an FPGA. We also hypothesise that deterministic concurrency control will be well suited for this task. This is because deterministic concurrency control is successful in distributed databases where the cost of communication between workers is high. Although we investigate into single node configurations, we anticipate that the cost of synchronisation between workers on an FPGA creates a similar parallel to distributed databases.

It is hoped that the outcome of this project will yield new insight into transaction processing on FPGAs, and provide the basis for future research into this area which has not been explored before.

1.2 Contributions

Transaction processing is at the heart of any DBMS. To the best of our knowledge, this work is the first research of transaction processing on FPGAs. The main contributions of this project can be summerised as follows:

- **An evaluation of the characteristics or transaction processing on FPGAs** - we have investigated what optimisations increase performance for processing transactions in-memory on an FPGA. This project also identifies the key bottlenecks in transaction processing on an FPGA of memory bandwidth and memory transfers between the CPU and FPGA device.

- **An investigation of concurrency control techniques for transaction processing on FPGAs** - from extensive research of concurrency control in transaction processing we identify deterministic concurrency control as a promising fit for FPGAs. We implement both this concurrency control scheme and more standard *two phase locking*, and demonstrate that deterministic is a better fit for the architecture of a FPGA.

- **A comparison between CPU and FPGA transaction processing** - we also implement both of these concurrency control techniques purely on the CPU to serve as a comparison with the FPGA based implementations. Overall, we show that the deterministic FPGA implementation out-performs the others, demonstrating the potential for the use of FPGAs in transaction processing.

1.3 Report layout

Chapter 2 introduces the current field of research in transaction processing. This will include a summary of popular concurrency control schemes used in modern DBMSs, and well as current research into using heterogeneous hardware in database acceleration. At the end of this chapter we will also discuss the details of the OpenCL programming model for FPGAs which we will use going forward in the report.

The core of the implementation work done for this project will then be discussed in detail in chapter 3. This includes the CPU implementations that we use as a baseline for our FPGA work, as well as the details of the FPGA designs and how they have been optimised to best make use of the hardware.

Chapter 4 will then evaluate the merit of the implementations. Firstly, we compare the CPU and FPGA based implementations individually to evaluate their performance characteristic. Then, we compare the overall performance of all of the implementations in throughput and latency.

Lastly, in chapter 5 we discuss the our overall findings from the project as well as the major challenges we faced during the project. Possible future extensions to the project are also presented that we did not have the time to explore during this project.

Chapter 2

Background

The following section aims to summarise the current research into transaction processing databases, and give an overview of the different techniques used in these systems. We initially outline the current methods of concurrency control used by distributed database systems to illustrate the possible approaches that could be used in the project. We then outline the current work going in to using heterogeneous hardware to accelerate database systems. Lastly, we discuss OpenCL and the use of high level synthesis languages in FPGA programming.

2.1 Concurrency control in database systems

ACID database transactions is an abstraction widely used in most database systems. It is highly popular as it abstracts away concurrent interactions between multiple commits being sent to the system by providing logical isolation between each commit, making the system easier for developers to reason about.

In order to maintain isolation between commits in an ACID database, a DBMS must guarantee a serial ordering of the transactions given to a system. The simplest way to do this would be to execute one transaction at a time, and wait until it is written to durable storage before starting the next. However, this will be slow as if one commit is waiting for disk read or write, or waiting on the network, all other transactions will also be held up. Furthermore, this would result in poor utilization of most modern multi-core systems and lack the ability to scale out across many servers.

Therefore a DBMS should be able to execute multiple transactions concurrently, while maintaining a serial execution order. Considerable care has been taken to perfect concurrency control protocols in database systems as communication overhead to guarantee ordering can often be the source of bottlenecks. The following sections aim to describe in detail the common concurrency control protocols used in database systems today.

2.1.1 Two phase locking

In 1976 one the first concurrency control schemes was introduced [7], which still remains popular today. This scheme is known as two-phase locking (TPL). The main idea behind this scheme is that each transaction must have a *growing* and *shrinking* phase. As in regular locking, a transaction must have a lock for each

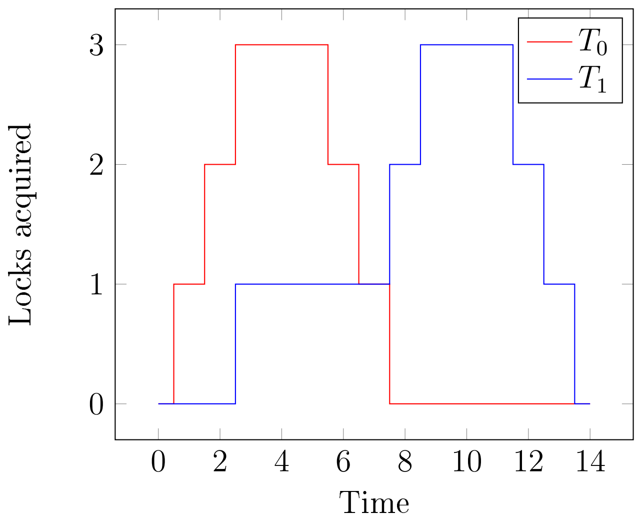


Figure 2.1: Diagram of two transactions executing with TPL. After obtaining one lock T_1 has to wait for T_0 to realise locks before it can gain more and execute.

record it accesses. During the *growing* phase, a transaction can request new locks. However once a transaction releases a lock, it cannot acquire a new one. Thus, each transaction must go through a *growing* phase, followed by a *shrinking* phase where all locks released with a period in the middle where the transaction is executed.

If two transactions have overlapping records, then the transaction that gets the overlapping locks first will stop the other's *growing* phase from ending and thus the transaction cannot start. Only when the first transaction has finished committing and enters *shrinking* phase is the second transaction able to acquire the rest of the locks it needs and continue. Thus, this ensures that two overlapping transactions cannot execute at the same time and therefore guarantees a serial ordering of commits. This is demonstrated in figure 2.1.

However, one problem with TPL is that conflicting transactions can result in deadlock in their growing phase depending on the order in which locks are acquired. There are a few ways that are commonly used to deal with this scenario:

- **Deadlock detection** - each transaction could be checked to see if its accesses cause a dependency cycle. If a cycle is detected, a transaction in the cycle can be aborted and run later.
- **No wait** - if a transaction requests a lock and it cannot be obtained then the transaction is aborted and all locks it was holding are released.
- **Wait-die** - each transaction is given a timestamp when it begins. If a transaction tries to acquire a lock that another is holding, it is permitted to wait on that transaction if it's timestamp is newer than the other transaction's. Otherwise, the transaction is restarted but keeping the original timestamp [2].

Although this fixes the problems with deadlock, TPL has problems with scaling due to *lock thrashing*. This happens as a transaction will hold all of the locks it needs until it commits, which blocks all other transactions that are happening concurrently that require any one of the locks that it holds. This causes problems when there is a large amount of transactions with high levels of contention and is the main bottleneck of all TPL schemes [38].

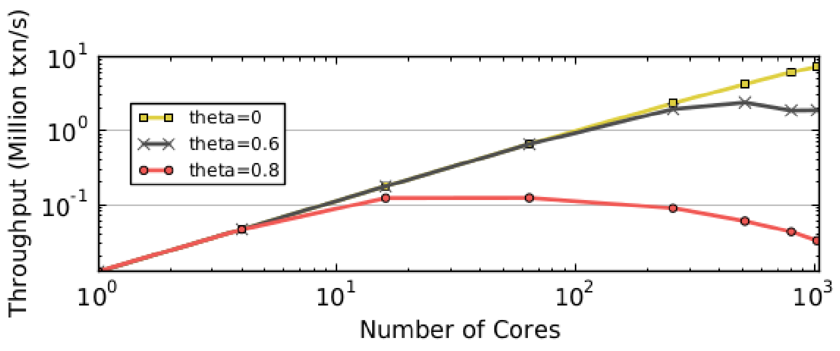


Figure 2.2: Results for a write-intensive YCSB workload using TPL with varying levels of contention [38]

2.1.2 Two phase commit

The problem when executing transactions that span multiple partitions (or replicas) of a database is that all parties involved in executing the transactions need to agree if it succeeded or not in order to maintain ACID. The classical way that distributed databases have been architected to solve this problem was pioneered by System R* [25] in the 1980s. This popularised two-phase commit - an atomic commit protocol for distributed systems. Two phase commit works by having an initial phase where all nodes try to execute the transaction. Then, in the second phase all participating nodes have to vote on whether the transaction succeeded or not. The problem with this methodology is that the second phase requires multiple network round trips between all participating nodes which can be the source of a major bottleneck in most systems. Moreover locks on records accessed for the transaction are still held when the protocol is taking place, further limiting performance.

Because of these problems, there has been a significant movement to relax some of the ACID guarantees in order to yield better performance and scalability. Amazon's Dynamo [6] relaxes consistency guarantees, while Google's Bigtable [4] completely removes support for multi-row transactions. This can be well suited to some applications. For example, in Amazon's case if reads of a shopping baskets contents from multiple replicas are inconsistent then the current contents of the basket must be the union of all the baskets [6]. However, a large number of applications do not fit these models well. In many applications database transactions mirror physical transactions, eg, money moving from one account to others or a warehouses stock being added to or taken from. If ACID properties are relaxed then serious issues can occur such as people spending money they do not posses or customers ordering items that do not exist. Moreover, relaxing ACID can often make it harder for developers to reason about their applications which can slow down development and cause bugs.

2.1.3 Deterministic CC

Traditionally, transactions processed by a DBMS are executed in a non-deterministic ordering. A DBMS will typically choose an ordering of transactions based on a number of runtime factors such as:

- Disk read times
- Hardware failure
- Thread scheduling

instead of processing transactions in the order given to the system. This stays within ACID guarantees, as it only stipulates that transactions must be processed in *some* serial ordering.

Consider an example database system. For each transaction executed in this system we first take out the locks on the tuples accessed by the transaction, then commit the transaction before releasing the acquired locks. If a transaction stalls due to deadlock or reading from storage, this can cause other subsequent transactions to be stalled as well. An example [30] of how this could be a problem is given below. Suppose three transactions were to be processed in order:

$$\begin{gathered} T_0 : \text{read(A), write(B), read(X)} \\ T_1 : \text{read(B), write(C), read(Y)} \\ T_2 : \text{read(C), write(D), read(Z)} \end{gathered}$$

If T_0 were to stall on reading X, T_1 would not be able to execute as T_0 would still hold a lock on B. If the system is deterministic, the commits would have to be executed in the order given and so T_2 would also not be able to execute as its read of C is dependent on the result of T_1. However, if we were able to process the transactions in a non-deterministic ordering, we could choose to process T_2 straight away and make better use of available resources. This would still maintain ACID as we are simply choosing a different serial ordering of transactions.

Figure 2.3 demonstrates this issue with deterministic concurrency control in an experiment [30]. In this experiment, each transaction accesses 10 of the databases 10^6 records. The time taken to process each transaction is relativity short, taking around $30\mu s$ from having all the locks to the last being released. Then, at 1 second a transaction acquires 10 locks and stalls for a full second. After the time has passed, it then commits and releases its locks.

While there is not a stalled transaction, both the deterministic and non-deterministic systems show similar levels of performance. Similarly both systems recover quickly after stalled transaction finishes and return to the usual levels of performance. However we can see that when the stalled transaction starts, the deterministic system yields much lower performance than its counterpart. In particular, since it cannot reorder its transactions, the throughput reaches almost zero at every level of contention as the stalled transaction clogs the system much like the previous example. Whereas, the non-deterministic system is able to still process transactions during this time as it can simply reorder the transactions and execute ones that do not conflict with the stalled transaction. Although it will slow down eventually if it runs out of transactions that do not conflict with the stalled transaction.

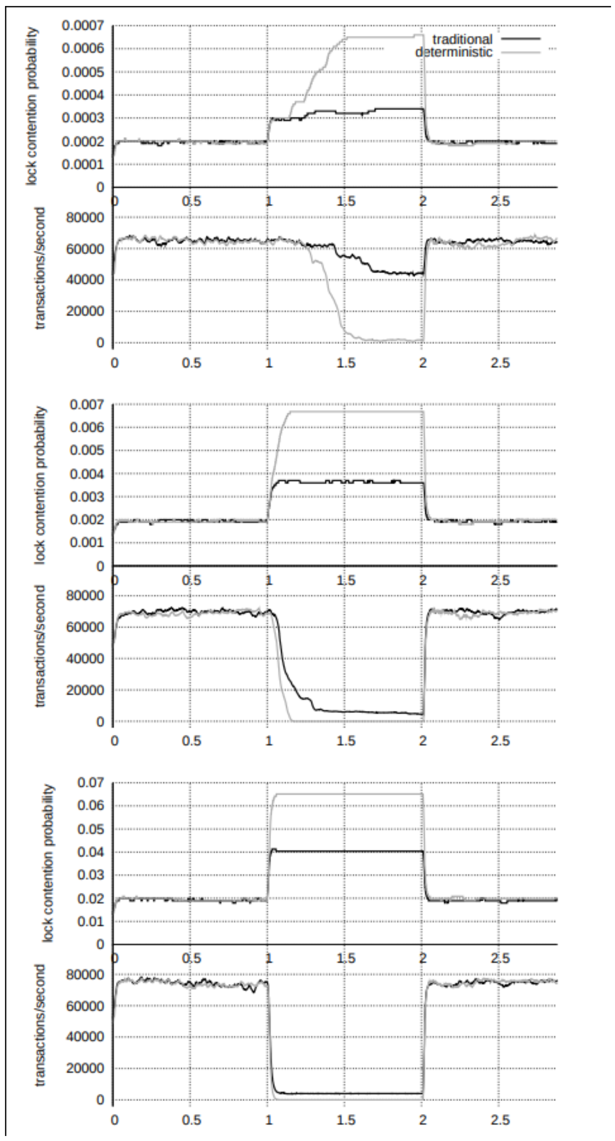


Figure 2.3: Measured probability of lock contention and transaction throughput with respect to time in a 3-second interval. Two transactions conflict with probability of 0.01%, 0.1% and 1% respectively [30]

From this evidence we can see that for any system that transactions frequently stall due to deadlock, disk read/writes or other problems, a non-deterministic approach will clearly yield much better performance. Thus, in the past most database systems have opted for this design.

In 2010, a paper was published proposing to revisit deterministic concurrency control [30]. This was motivated by changes in the landscape:

- **Increased performance in processing transactions** - with the increase in availability of main memory many applications can now fit entirely in main memory, and larger applications can store data on fast SSD based storage. This combined with the continued increase in reliability of hardware means that the probability of transactions stalling is greatly reduced.

- **Rise of OLTP workloads** - OLTP systems typically process short-lived and repetitive transactions that access a small number of records at a time [19].

This greatly reduces the chances of a stalled transaction clogging the transactions behind it.1

These two factors greatly reduce the frequency of transactions stalling and causing clogging in a deterministic system. Therefore in practice we are unlikely to see this worse case performance seen in Figure 2.3. However, since the transactions are executed in a deterministic ordering there is no need for communication between distributed threads or nodes executing transactions. Thus we can completely eliminate the overhead of communication between distributed transactions seen in non-deterministic systems as they scale horizontally.

This idea was fully realised by Thomson et al. in Calvin [31] - a transaction scheduling and data replication layer making use of deterministic concurrency control. Calvin provides a sequencer layer that is capable of taking in transactions from clients and maintains an agreed ordering of transactions across all replicas in the system. This is done by either having a master sequencer that forwards batches of order transactions to all other slave replicas, or by reaching consensus on a ordering via the Paxos algorithm [22]. The scheduling layer then obtains locks for each transaction, strictly in the ordering given by the sequencer layer. Then, when all locks are obtained, the transaction is sent to a worker thread for execution. Another key innovation of Calvin is that it solves the problem of using slow storage that is problematic for determinism as we have discussed before. When the sequencer receives a transaction that could incur a disk stall, it waits and sends a request to storage to obtain the relevant records. This then reduces the possibility of a disk stall while a transaction is holding locks, thus further reducing the chances of clogging.

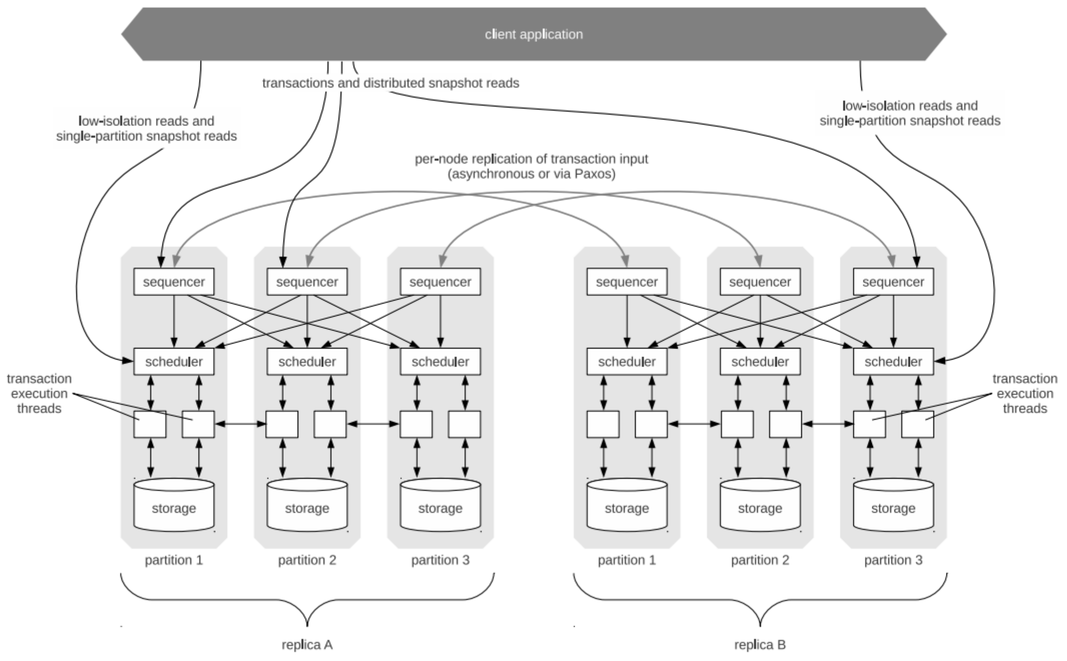

client application
transactions and distributed snapshot reads
low-isolation reads and single-partition snapshot reads
low-isolation reads and single-partition snapshot reads
per-node replication of transaction input (asynchronous or via Paxos)
sequencer
scheduler
transaction execution threads
transaction execution threads
storage
partition 1
partition 2
partition 3
partition 1
partition 2
partition 3
replica A
replica B

Figure 2.4: The system architecture of Calvin [31]

The combination of these design decisions allows Calvin to scale well compared to other systems using other concurrency control schemes. In 2017, Harding et al.

compared Calvin to a number of popular concurrency control schemes in a number of benchmarks [11]. To do this they implemented their own framework called Deneva which could be adapted easily to multiple concurrency control protocols to compare their bottlenecks.

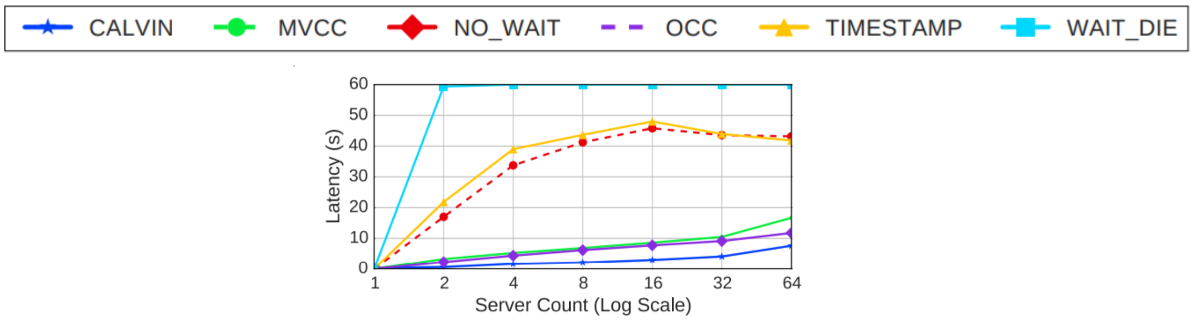


Figure 2.5: 99%ile Latency from a transactions first start to its final commit for varying cluster size [11]

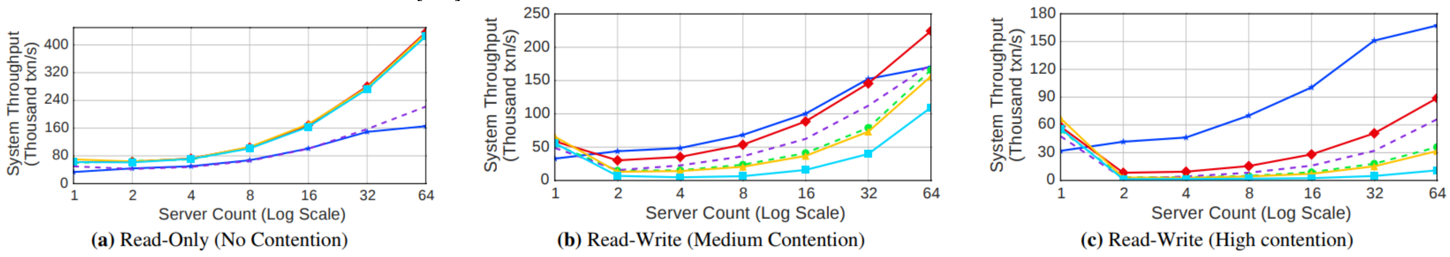


Figure 2.6: Throughput for the protocols using variations of the YCSB workload and different cluster sizes [11].

As can be seen by Figures 2.5 and 2.6, Calvin typically scales horizontally well compared to other schemes. In particular, the latency of transactions remains extremely low and scales with the log of the server count. This is due to the property of not having to perform many network round trips in order to acquire locks as is necessary in a non-deterministic setting. Furthermore, Calvin manages to have a similar throughput no matter the level of contention in the transactions, due to locks only being held purely when the transaction is being executed. The downside of Calvin's design is that the scheduler that gives out the locks to each transaction is single-threaded in order to acquire the lock in the given serial ordering. This means that when there is a read-only workload, it is not able to keep up with the other protocols.

Overall, deterministic concurrency control is a highly promising scheme that offers large improvements in performance over the more traditional two phase commit when scaling horizontally.

2.2 Hetrogeneous hardware in database systems

Historically, the steady increase of transistors in CPUs over time from Moore's law has meant that CPU performance has increased steadily over time. However, this is no longer the case. While the number of transistors and core count in modern CPUs are still increasing, single threaded performance improvements of CPUs has slowed down over the last decade [28]. While more cores allows us to better exploit parallelism in applications, due to Amdahl's law, even if most of an application can exploit parallelism it is still capped in potential speed-ups by its serial parts. However, the amount of data we need to process is still growing exponentially. In order to deal with this, modern data centres are becoming more heterogeneous to keep up with demands. In particular, the use of FPGAs, ASICs and GPUs have become extremely prevalent in many areas including finance, machine learning and data processing. The following section aims to outline the current work in using heterogeneous hardware to accelerate database systems

2.2.1 FPGAs

FPGAs (field programmable gate arrays) are chips that are able to be re-programmed in order to emulate almost any physical hardware circuit (within it's size constraints). FPGAs are typically slower than ASICs (application specific integrated circuits) as their reprogram-ability means that the circuitry will not have as optimal of a layout and thus result in a poorer clock speed. However, FPGAs are off the shelf components and so are *significantly* cheaper than their ASIC counterparts and have the flexibility to be reprogrammed and updated down the line while still yielding similar performance increases to ASICs over CPUs.

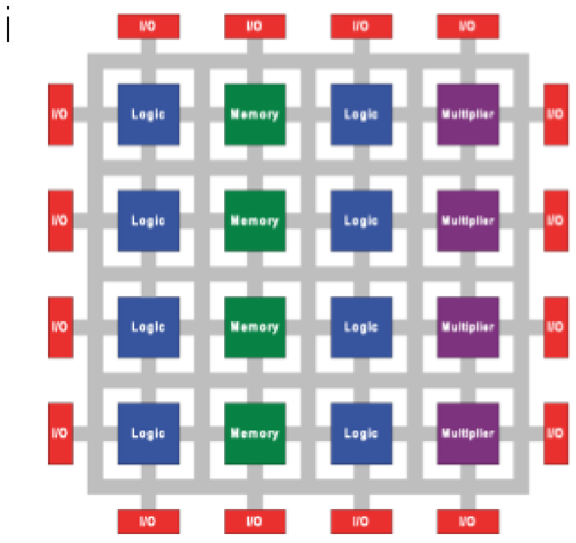


Figure 2.7: The basic structure of an FPGA [21]

FPGAs consist of a number of LUTs (small look-up tables), on-chip memory, and dedicated hardware (such as multipliers, MUXs etc.) that are connected by a fabric that can be re-programmed to connect these components together in different configurations. The advantage of using an FPGA over a CPU is that specific functions can be directly written into an FPGA and deeply pipelined allowing for much greater throughput than is possible on a CPU. Furthermore, multiple functions can be placed in parallel on a FPGA (much like a SIMD processor) allowing far greater levels on parallelism than is possible even on a large core count CPU. Because of these benefits, there has been a large amount of research into using FPGAs to accelerate database systems.

In 2014, Ibex [36] was introduced showing that parts of SQL query could be off-floaded to an FPGA. Ibex is an *intelligent storage system*, sitting between a DBMS (MySQL in its case) and its storage. All data requests from the DBMS are routed though the FPGA. Then the FPGA reads the relevant tuples from storage. Before handing them back to DBMS, Ibex can then apply a number of operations to the tuples - projections, selections and `GroupBy`. These operations can be configured by the DBMS via a register file on the FPGA. If parts of the query cannot be off-loaded to Ibex, the tuples can simply be passed through back to the DBMS unchanged. Compared to a common MySQL storage engine MyISAM, Ibex offers greater performance at a lower power consumption. Ibex also shows that an FPGA system can easily be plugged into an existing DBMS and provide significant hardware acceleration.

Simple in-memory database systems are becoming increasingly popular in web backends to cache results to common database queries to increase responsiveness in delivering pages. Examples of these are Redis and Memcached. In 2013, it was shown that FPGA could be used to implement a key-value store [3] that could be orders of magnitude faster than its CPU based alternatives at this task. The design was highly pipelined, and could handle instruction level parallelism. At its peak performance, it could process over 13 million requests a second compared to under 2 on a 8 core CPU. More impressively it could achieve this performance at a lower power consumption to the CPU, processing 106K requests per second per watt compared to the CPU's 7K. This then proves the viability of in-memory data storage on an FPGA, which is key to an in-memory transaction processing system succeeding on an FPGA.

However, to the best of our knowledge, there has not been any published research into transaction processing on FPGAs despite seeing the benefits of using FPGAs in database systems seen in this section.

2.2.2 GPUs

GPUs (graphics processing units) are processors that feature a massively parallel architecture of basic processing cores. Typically they have a far greater memory bandwidth than CPUs, and have a higher peak FLOPs. Although these processors were traditionally designed to accelerate 3-D graphics in desktop systems, they have programmable support for compute via support for languages like CUDA and OpenCL and are widely used in applications that can take advantage of their parallel architecture such as machine learning. There has also been a large amount of research into using GPUs to accelerate various aspects of database systems.

GPUTx [12] was proposed in 2011 - an in memory transaction processing system running completely on a GPU. This paper particularly demonstrates the performance gains that a massively parallel architecture can give to a transaction processing system. When running on a single GPU core, GPUTx only achieves the throughput of 25%-50% of a single CPU core. However when utilizing the whole GPU, GPUTx reaches 4-10X the performance of a CPU implementation. While this paper clearly demonstrates the possibilities of GPUs in transaction processing, it does not bring any novel ideas to the table. GPUTx is implemented by bulk processing transactions based on current concurrency control method (such as TPL and K-Set). This paper demonstrates the advantage of using accelerator devices in transaction processing however the techniques used may not be applicable to transaction processing on an FPGA because of the large differences in the hardware architecture.

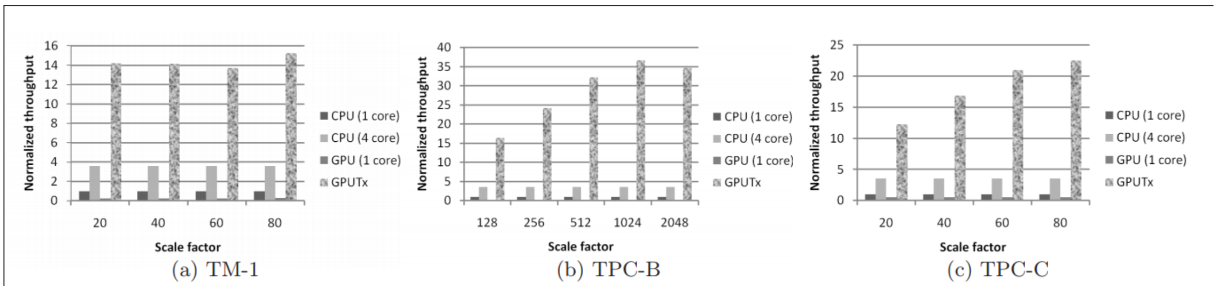


Figure 2.8: Normalized throughput for a number of benchmarks comparing GPUTx to CPU implementations [12]

2.3 OpenCL

Open Computing Language (OpenCL) is an open parallel compute framework for writing programs that are able to run across heterogeneous hardware including CPUs, FPGAs, GPUs and many more hardware accelerators. OpenCL is implemented by many companies for their devices including Intel, AMD, NVIDIA, Xilinx and many more [29]. Code for the hardware accelerators can be written in either C99 or C++11 syntax, with a few restrictions and additional tags and keywords. The program interacting with the accelerator can then interface with it using a provided API.

The OpenCL programming model is split into two parts - the *host* and the *device*. The device incorporates what is run on the hardware accelerator. Regardless of the specifics or how the device works (whether GPU or DSP) the model remains the same. Firstly, *kernels* are the programs are run on devices. They are simply functions that are run till completion and can have a number of arguments given to them of pointers to buffers in memory. Kernels can either be implemented in a task parallel or data parallel way. In the task parallel way, a kernel simply executes a fixed function and many of these tasks can be executed in parallel. In the data parallel way, a global group of work is assigned to the device. This is then split evenly into work groups, each consisting of a number of individual work items. The kernel is then run for each item in every work group. The kernel query its local work item id as well as its group id at runtime and perform a different function based on its ids. A way this is typically used is if the kernel is performing a operation on groups of vectors, the local work id could be used as an index on the vector is group is working on. Multiple work groups can then run in parallel across many *compute units* (typically cores) on the device at runtime. However, the device has control over scheduling work onto its compute units and not all work groups are guaranteed to be running concurrently.

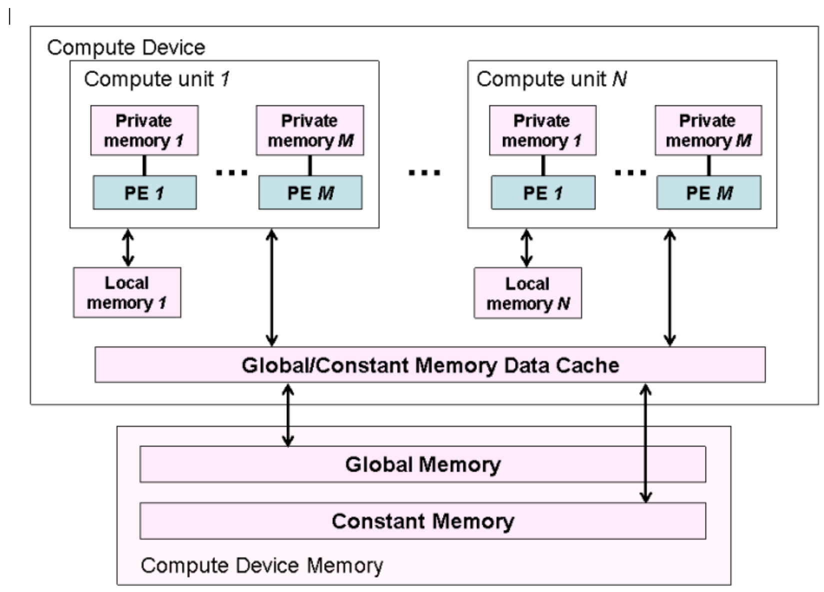


Figure 2.9: OpenCL compute device architecture [26]

The host then has the job of orchestrating the whole operation. First, it queries the devices available to it, and selects one to use. It can query various parameters about the device at this time to know the type of device and what it supports. Then the host has to re-program the device with a binary of the kernel it wants to run on it. Once the kernel is programmed to the device, the host then can set up buffers that will map to the kernels arguments and be used for the data transfers to and from the device. The host can then write the arguments to the host's memory. Following that, the host can then issue a command to the device to execute the kernel. The host can then wait for the kernel to finish executing, or issue more commands to be executed on the command queue. When the kernel is done executing, the host can then transfer any results from the device back to the host.

Programming OpenCL for FPGAs is mostly the same as any other OpenCL device, however there are some slight differences. First of all instead of the device being programmed with a binary by the host, it must be programmed with a bitstream. Since this bitstream is a physical layout of the hardware for our specific target card, you cannot run your kernel on any other different FPGA device. Furthermore, this means that multiple kernels cannot run on the device unless they were compiled together. Secondly in order to reprogram the device with a new kernel, it must be already running an OpenCL kernel. This meant that when we the card was initially configured, it needed to be bootstrapped with an OpenCL kernel. This was achieved by writing a OpenCL kernel's bitstream to the card's on board flash memory via a USB JTAG interface. Then whenever the card is rebooted, it automatically loads up the bitstream on the flash memory, and therefore is able to be reprogrammed with an OpenCL kernel.

An alternative to using OpenCL to program FPGAs would be to use a *hardware description language* (HDL) such as Verilog or VHDL. These languages are traditionally used for hardware design and give much finer grain control over the hardware that is synthesised. However, this also means that it is much more complex and time consuming to write programs in these languages. Thus, a decision was made to use OpenCL for the implementations as although this could lead to a less optimal design for the FPGA, the aim of this project is to investigate what methods of concurrency control fundamentally work well for transaction processing on an FPGA. Therefore, relative performance of the methods to each other is more important an the absolute performance that is achievable and so the ease of programming with OpenCL is a great asset. It can also be noted that the debugging designs is much easier in OpenCL, as designs can be run in an emulator and debugged like a C program. Whereas with a HDL language, you have to write test benches for the design and debug the signals in the emulated hardware.

Chapter 3

HOBBES

In this chapter we present Hobbes, our implementation of transaction processing using deterministic concurrency control on an FPGA, as well as our implementation of TPL on the FPGA and CPU implementations. Section 3.1 covers the applications we support across all the implementations. The details of our CPU based implementations is then covered in section 3.2. Finally, the details of the FPGA based implementations as well as the design considerations for maximising the performance of the FPGA kernels are outlined in section 3.3.

3.1 Target applications

For this project, it was decided to use the Yahoo cloud scalability benchmark (YCSB) [5] as the main target application. YCSB was developed out of the need for a standard benchmark for modern document stores that do not support a strong relational model. Unlike traditional industry benchmarks in transaction processing like TPC-C [32], YCSB does not try to emulate a specific application or use-case. Instead, YCSB defines a simple set of *"CRUD"* (create, read, update, delete) operations that a database must support and then defines a variety of workloads with different proportions of these operations. This then enables YCSB able to show a wide range of possible workloads that are interesting for many different applications.

The operations that YCSB defines are as follows:

- **Insert** - Insert a new record. [5]
- **Update** - Update a record by replacing the value of one field. [5]
- **Read** - Read a record, either one randomly chosen field or all fields. [5]
- **Scan** - Scan records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen. [5]

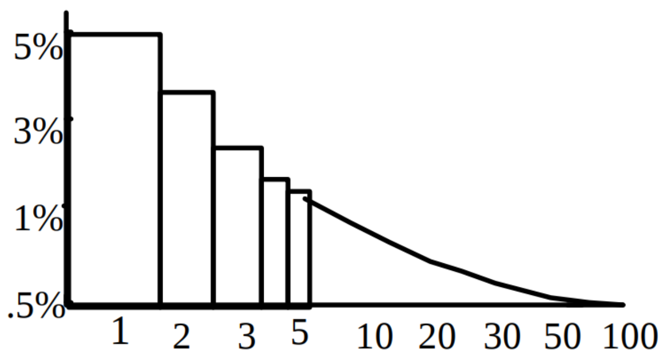


Figure 3.1: The probability distribution of a Zipfian distribution [9]

A key aspect of the YCSB benchmark is the distribution of keys in the transactions. The main distribution of keys for the transactions in YCSB is the Zipfian distribution which can be seen if figure 3.1. Each integer k in the distribution has weight proportional to $\frac{1}{k}^{\text{theta}}$. This means that some keys will be very popular, while others at the other end of the distribution will be very sparsely accessed. Furthermore, the parameter theta can be increased or decreased to vary how skewed the distribution is towards the popular keys. This is particularly important when assessing concurrency control, as it allows the ability to vary the level on contention on the most popular keys.

YCSB defines 5 workloads as follows:

Workload	Operations	Record selection	Application example
A Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere
D Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E Short ranges	Scan: 95% Insert: 5%	Zipfian / Uniform	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

Table 3.1: The YCSB workloads as defined in [5]

YCSB was selected as the main application for this project for a number of reasons. Firstly, being able to easily vary the contention on keys with the parameter theta allows an exploration of the relative performance of the different methods of concurrency control under varying circumstances. Furthermore, having to only support a simple "CRUD" API means that the FPGA designs can be relatively simple and make it easy to identify and focus on bottlenecks. Lastly, YCSB is a popular benchmark used to evaluate many transaction processing systems and allows us to easily compare how our implementation performs against other systems in the current landscape.

YCSB also provides a YCSB client. This is a Java application set up to run the 5 workloads on a connected database and collect statistics such as throughput and latency of transactions. The client is designed to be extensible, and new databases can easily be added by a new *database interface layer*. This layer provides the setup for the connection to the database as well as an implementation of all of the functions in the CRUD API needed for YCSB. However, we opted not to use the YCSB client for benchmarking our system. This is because traditionally the YCSB client would be connected to a multi-node database with a connection over an interface like TCP. However, since the project is focused purely on a single server setup, the overhead for any connection and translation to the client is much more significant and could lead to an unwanted bottleneck. Furthermore, by implementing the benchmark ourselves we are able to have much finer control over the metrics we record.

3.2 CPU based implementations

The first part of this project was to create a two phase locking based system, as well as a deterministic concurrency control based system based purely on the CPU to establish a baseline for performance.

For both types of concurrency control we tried to keep the architecture constant, barring the mechanisms used to control the concurrent execution of transactions. Common to both, we have a *application* that is responsible for generating the transactions used to benchmark the system. We then have a *scheduler*, which schedules work to the workers and monitors the workers for finished transactions. *Workers* are responsible for executing transactions through a shared in memory table. Transactions are given to each worker by the scheduler via it's own individual work queue. Each worker also has an outgoing queue shared with the scheduler to queue the results of each transaction to. Workers are implemented as C++ threads, and simply spin waiting for work to be given to them.

For our experiments we measure throughput by recording the time taken to execute a large number of transactions. This is measured from the time taken before the scheduler schedules the first transaction, to the time that the scheduler receives the result of the last transaction to finish. To ensure that we do not incur any overhead by generating transactions during our benchmarking, we generate all of the transactions in a benchmark before the start and store them in memory.

To gauge where bottlenecks are in our implementation we used Intel's software profiling tool, VTune [16]. Vtune enables us to run our application, profiling data about its execution such as time spent executing functions and thread utilisation. Using this tool, we found that allocating memory during runtime was a major overhead. To get around this we ensured that all data in transactions was fixed size and so did not have to be allocated during the benchmark. In addition to this instead of allocating new result objects when we finish a transaction in each worker, we add to the results queue a pointer to a result in a circular results buffer.

For increased efficiency, we also implement *read-write locking* for both methods of concurrency control as YCSB benchmarks typically include a mix of read and write transactions. Read-write locks maintain the following invariants: (a) a read lock is acquired iff there exist no writers, (b) a write lock is acquired iff there exist no readers or writers. This is a effective optimisation for locking as it still guarantees isolation of the transactions, but will mean that multiple readers to the same tuple will not experience contention.

3.2.1 Deterministic

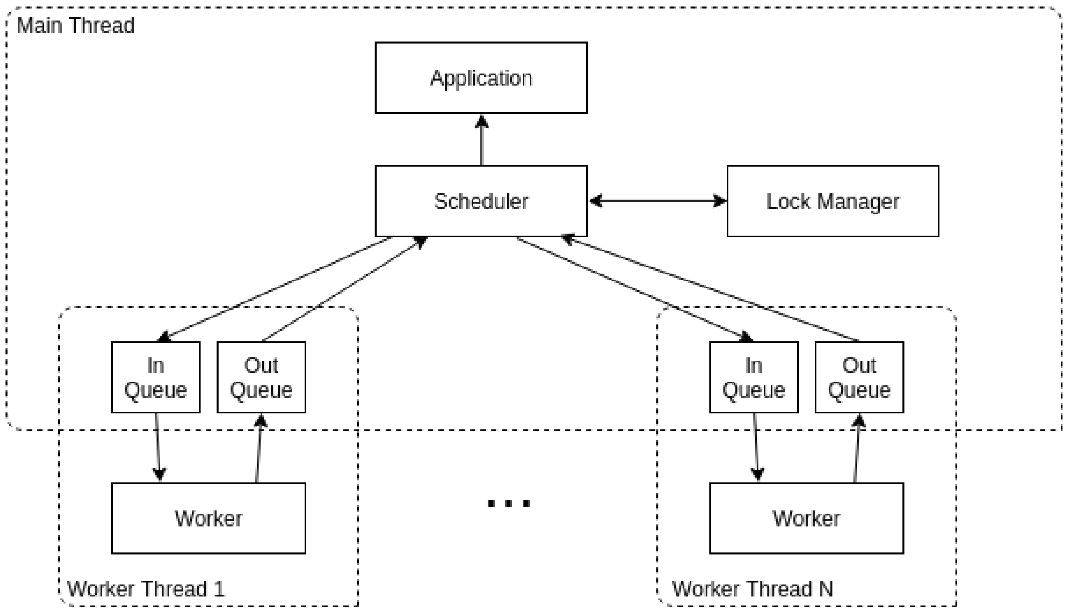


Figure 3.2: The architecture of the deterministic concurrency control implementation

In the deterministic implementation, we tried to keep the code faithful to the implementation used in Calvin [31]. The main alteration we make is to remove the sequencer, as since we only executing on a single node, the order in which the scheduler receives transactions is the only possible ordering.

A key aspect of the deterministic implementation, is the *lock manager*. The lock manager implements read-write locking per tuple in the table. Every transaction must be processed by the lock manager before it can be processed by the scheduler. When the scheduler processes the next transaction in the queue, it initially attempts to lock it with the lock manager. If the transaction is locked successfully, the transaction is scheduled to a worker. Else, the scheduler moves on to the next transaction in the queue. Then, when a worker completes a transaction, the scheduler unlocks the transaction using the lock manager. The lock manager then returns a list of transactions that have been made ready to execute by the lock(s) being released. The scheduler then immediately schedules these transactions to workers.

In order to maintain determinism, the lock manager must keep an ordering of the transactions waiting on locks on the tuples. This is done by keeping an ordered list of the transactions waiting on each tuple, along with whether it is a read or write access. Then the transactions that are granted the lock by a lock that has been released can be determined by the following algorithm:

Listing 3.1: Algorithm for granting locks adapted from [37]

```
if (realeasedRequest == requests->begin() &&
    (realeasedRequest->isWrite ||
      (!realeasedRequest->isWrite && it->isWrite))) {
  // If a write lock request follows, grant it.
  if (it->isWrite)
    newOwners.push_back(it->txn);
  // If a sequence of read lock requests follows, grant all of them.
  for (; it != requests->end() && !it->isWrite; ++it)
    newOwners.push_back(it->txn);
} else if (!ongoingWrites &&
            realeasedRequest->isWrite && !it->isWrite) {
  // If a sequence of read lock requests follows, grant all of them.
  for (; it != requests->end() && !it->isWrite; ++it)
    newOwners.push_back(it->txn);
}
```

3.2.2 TPL

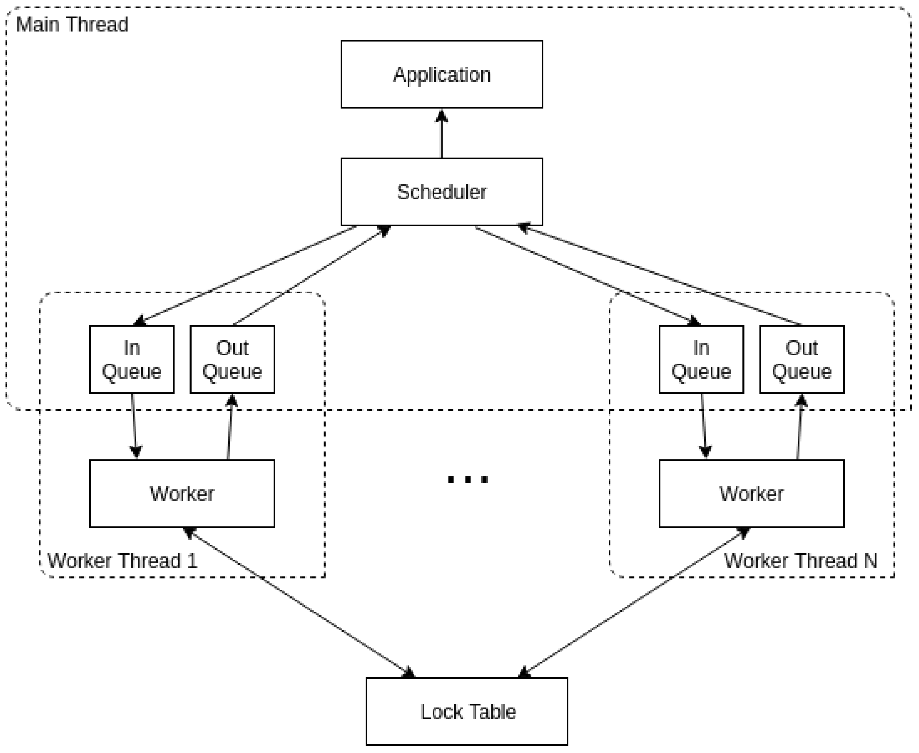


Figure 3.3: The architecture of the TPL implementation

For the two phase locking implementation we remove the lock manager from the scheduler, and simply schedule work to the workers as needed. Then to manage concurrency between the transactions, each worker thread shares access to a *lock table*. This lock table implements a read-write lock per tuple (similar to our deterministic lock manager), and is implemented through C++'s standard library `shared_timed_mutex`. Thus, if there is any contention on a tuple, one worker always proceeds while the other threads are blocked. Deadlocking can also be avoided by ensuring that workers obtain the locks that they need in a deterministic ordering.

3.3 FPGA implementations

3.3.1 FPGA design considerations

FPGAs as detailed in section 2.2.1, are hardware circuits that are able to be reprogrammed with *bit-streams* to emulate physical hardware. In section 2.3 we then discussed the exciting new development to write programs for these devices in high level compute languages like OpenCL that allow us to compile more tradition *"C like"* programs into *bit-streams* for our FPGAs. However, we found that simply compiling our OpenCL programs for FPGAs without taking into consideration the synthesis of the programs into hardware, will not generally lead to good performance or utilisation of the hardware. In this section, we discuss the design considerations that we need to take when porting the earlier CPU experiments onto using the FPGAs.

Pipelining

The first important concept to consider is *pipelining*. This is the concept of a hardware design consisting of many sequential steps that instructions or data must flow through to finish their computation. When one instruction finishes a step, it can then move on to the next step, allowing the next instruction to then compute that step. This leads to *pipeline-parallelism*, where multiple instructions can be executing concurrently in the pipeline at different stages. This can a big performance advantage when using an FPGA over a general purpose processor as multiple instructions are in flight at a time, a much lower number of cycles to output a result can be achieved. For example Sirowy Et Al, found that their fully pipelined design for JPEG image compression could compute one color component per cycle - 1155X faster than their general purpose CPU implementation [27].

However, it is not always possible to create a perfect pipeline in which the next instruction is able to execute immediately after the previous. A common example of this is an external memory operation, as the time taken for this operation can vary (especially if caching is used). If this operation takes longer than the time allocated to it, it will effectively *stall* the pipeline whereby all instructions behind it have to wait. This then reduces the effective parallelism in the pipeline as stalls in the pipeline create *bubbles* of empty instructions [13].

There are a few ways in which this was taken into account in the FPGA designs. Originally, we branched into different pieces of code to execute the transaction based on it's type. However, since each piece of code could have a different amount of execution time, the compiler can only pipeline up to the point where the code branches and execute the branch as a single step in the pipeline. Thus, this removes any pipelining from the design. Therefore, we made sure to make each transaction execute in one main loop rather than have separate loops based on the type of transaction. This then allows each loop iteration to be pipelined, so that multiple transactions can be executing concurrently in the design. This is possible as in

YCSB all columns are of a fixed size, and so we can make each transaction simply execute over all of the fixed bytes of a column in the store. If we want to support other transactions such as full row reads we can simply construct this as multiple read transactions.

Read and write dependencies within a loop can also cause issues for the compiler being able to effectively pipeline a design. If we load or store to global memory inside of a loop, the compiler will make sure that the next iteration of the loop is not run until the load or store is completed to ensure correctness if there are dependencies on the memory in future iterations of the loop. However this limits the performance of the design, as we are stalling the pipeline on these memory actions. This was a big problem for the performance of our FPGA designs as the bulk of our kernel is reading and writing to memory within a loop. To avoid this, the Intel OpenCL compiler provides the `ivdep` pragma which instructs the compiler that any loop this pragma is placed above does not have memory dependencies between loop iterations. This then means that the compiler can allow a higher rate of instructions going through the loop as we do not have to stall loop iterations on memory accesses.

Memory accesses

Another aspect we must take into account when writing programs for an FPGA is the way in which memory is accessed. This is particularly important on an FPGA as we do not have the benefits of features large caches and speculative execution which benefit memory performance on a general purpose processor. Thus, to maximise memory bandwidth we must design our programs memory accesses to compliment the DRAM interface.

One important optimisation is alignment. If we do not our align structures with the width of the memory interface then we will end up adding additional reads or writes to where the data overlaps the memory interface width, reducing our effective memory bandwidth. Our particular hardware needs 64-bit memory alignment. To get around this, we make sure to pad all of our structures to 64-bits so that when the design indexes a structure in a memory buffer, it's address will always be a multiple of 64-bits. Furthermore, when transferring data two and from the device, we must ensure that the buffers are 64-byte aligned to allow direct memory access (DMA) with the host and board.

Since memory interfaces are typically very wide, performance can also be increase by coalescing reads and writes to memory. In our design we copy each transaction object into local on chip memory at the start of each transaction and then copy the result object into global memory only at the end of the transaction. Doing these memory operations in bulk, rather than reading or writing to individual fields during the transaction ensures that the reads and writes are coalesced and so the compiled hardware can make full use of the width of the memory bus.

Scaling up

Once we have maximised performance from a single pipeline design, we need to scale up the design to make full use of all of the FPGAs resources. One way this could be achieved is to simply compile the design with multiple identical kernels. However this would mean that we would have to manage in and out buffers for each kernel which would lead a higher number of smaller data transfers between the host and device leading to poorer usage of the PCIe bus. To get around this problem, Intel's compiler provides two different methods - *kernel vectorisation* and *compute units.*

For both of these methods, we need to use *work groups* rather than execute all of the transactions in a single batch. Instead of iterating over all of the transactions, we call `get_group_id` and `get_local_id` to get the local and group id of the work item which we can then use to calculate the index of the transaction. The compiler then creates the pipeline to iterate through the work items in a work group.

Kernel vectorisation works by creating extra hardware that executes multiple work items in parallel, much like a single instruction multiple data (SIMD) processor. The compiler does this by creating extra hardware for each scalar operation in the kernel, so multiple work items can be executed in parallel. On the other hand, telling the compiler to create multiple compute units for a design simply creates multiple unique parallel pipelines. Additionally, this creates a hardware scheduler on the FPGA which schedules work items onto the compute units during runtime.

While both methods increase the amount of parallel work at the expense of hardware resources, kernel vectorisation is more efficient, as the parallel memory accesses are able to be coalesced and so leads to better utilization of memory bandwidth [13]. However, since transactions are independent and access random memory it is hard to match them to a SIMD workflow. Moreover, kernel vectorisation does not allow work id dependent branching which makes it difficult to use as we need to branch based on each transactions type (if it is a read or write etc...). Thus to scale up the designs in this project we use multiple compute units.

Memory transfers

Data is transferred to and from the FPGA via *direct memory access* (DMA) over the PCIe bus. DMA is an I/O acceleration technology that allows hardware devices to access the CPU's main memory, without independently of the CPU. In order to initialise a DMA, the CPU must perform some initialisation as well as handle an interrupt when the transfer is finished. This overhead makes small memory transfers inefficient as demonstrated in the experiment below.

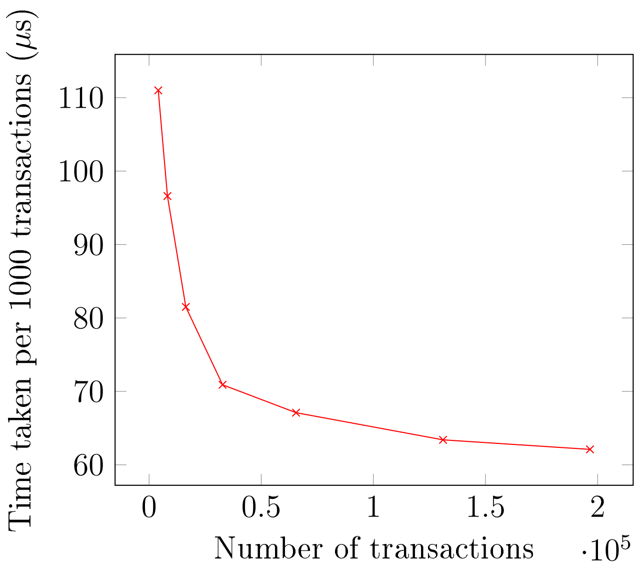


Figure 3.4: Time spent transferring the transaction objects to the FPGA (via DMA on the PCIe bus) per 1000 transactions, against the number of total number transaction objects transferred in one batch.

As seen in figure 3.4, the more transaction objects we sent in a batch the larger the total memory transfer size resulting in a slower effective time to transfer each object as the overhead of DMA is amortised. However as the size of the batches is increased further, the more this payoff is reduced.

Therefore, we must ensure that the size of the batches processed on the FPGA are large enough so that the overhead of DMA does not cause the memory transfers to dominate the kernel execution time.

3.3.2 Processing transactions on FPGA

Like in our CPU experiments, we have tried to keep the architecture similar for our TPL and deterministic implementations on the FPGA. We show this architecture in figure 3.5 below:

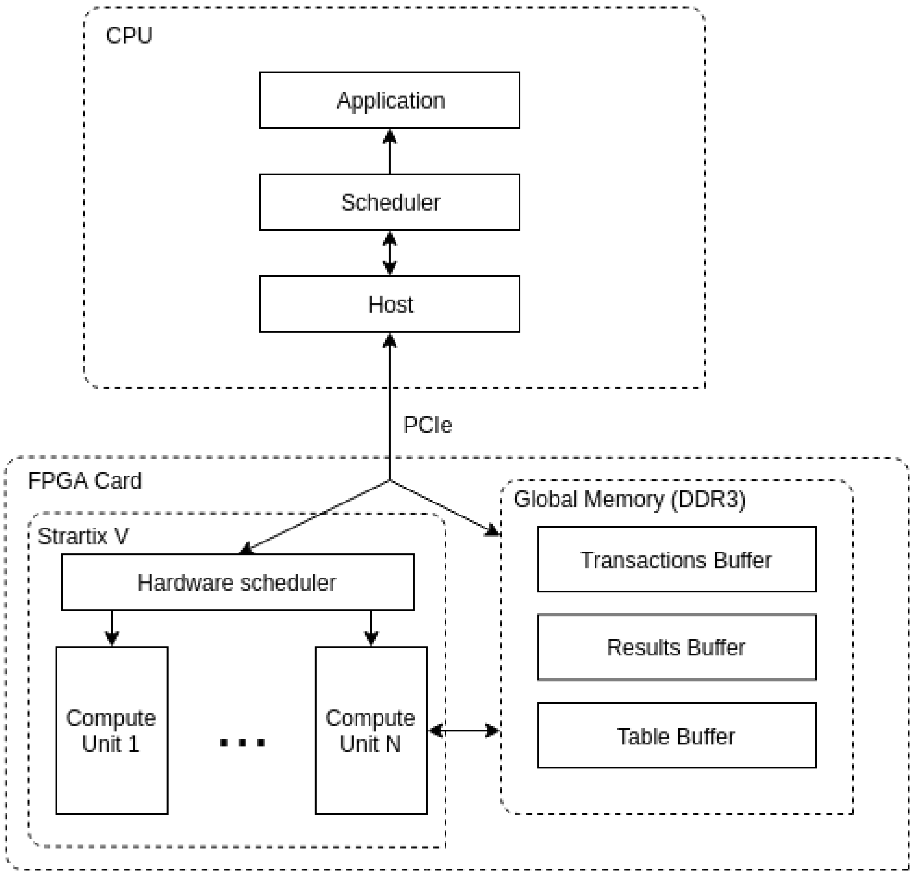


Figure 3.5: The architecture of transaction processing on the FPGA.

As in the CPU implementation, we also have a scheduler that has the role of managing the execution of transactions in the system. However, instead of scheduling individual transactions to be executed, the scheduler must now lock batches of transactions as we must be able to saturate the kernel's pipeline with transactions on each execution. As demonstrated in figure 3.4 previously, it is most efficient for memory transfers if we transfer large batches of transactions at a time. Batches of 131072 transactions were chosen as the speed-up beyond this in figure 3.4, and so makes a good trade-off between transfer speeds and the overall latency to complete transactions.

When the scheduler has filled a buffer to be executed, it passes it on to the *host*. The host is responsible for managing all aspects of the FPGA card. When the program is started, the host programs the FPGA with our kernel's bitstream, allocates the three buffers in the devices global memory and assigns pointers to them as the kernel's input arguments. The transactions buffer acts as the input to our kernel, and consists of `structs` containing the data needed to execute each transaction. The results buffer acts as the output of the kernel and contains result `structs` containing the result of each transaction. Lastly, the table buffer allocates the area in memory to be used for the data storage. When the scheduler gives transactions to be executed to the host, it writes the new transactions to the transactions buffer and queues a kernel execution to the command queue. Then, when the scheduler is finished filling the next batch of transactions it calls the host for the results of the previous batch. When this happens, the host waits for the execution of the kernel to finish, and then reads the results buffer from the FPGA's memory. The host blocks until all of the results are read.

For each transaction, the kernel takes the following steps. Firstly the kernel gets its work group id and work item id to calculate the unique index of its transaction in the transaction buffer. Then, we buffer the transaction object into private memory. Branching on the transaction type, we either preform a read or write to the table in global memory with the key the value given in the transaction object. Lastly, we write the result of the transaction to the transaction buffer.

Challenges

A large problem that we had when writing the kernel, was the speed of the accesses to global memory. From viewing the profiling data from the kernel execution all memory accesses were getting 3-8% utilisation of the memory bandwidth. In order to improve this we investigated into the hardware the compiler was generating for these memory accesses.

From looking at the report that Quartus had generated from the compile, we found that the compiler had generated *non-aligned burst coalesced* LSUs for the loading of transaction objects and the storing of result objects. Non-aligned memory accesses can cause very poor memory performance as extra reads or writes have to be performed where overlaps in the memory word size occur. Although we were 64-bit aligning the transaction and results structs, the compiler was still generating misaligned LSUs because it was not aligned to the memory word size of 512-bits. To get around this problem, we forced these structs to be 512-bit aligned. Since the size of each field is 512-bits, this forces both structs to be 1024-bits in total size. This resulted in compiler generating *streaming* LSUs for the memory accesses.

Next we looked at the hardware generated for the memory operations on the table. Initially in the design, we looped through the reads/writes, reading/writing to each byte of the field sequentially. Thus, we were relying on the LSU to buffer and coalesce the sequential bytes in order to ensure efficient use of the memory bus. However, this was clearly not the case from the profiling data. To solve this, we created a struct called `Col` which only contained a fixed size array of the size of a field in the table. Then, in the kernel we fully removed the loop and simply perform the read/write operation as a assignment of a `Col` struct. This forced the compiler to generate an LSU of exactly the width of a field, so that each read or write transaction uses accesses exactly one line of memory in one go.

Figures 3.6 and 3.7, show the flow diagrams before and after these described changes. As can be seen, these changes result in a far simpler flow through the kernel, and removes the non-pipelined area (shown in red) where we perform the transaction. The result of these improvements pushed the utilisation of the memory bandwidth for each memory access to 100%, and resulted in an around 5X improvement in overall kernel throughput.

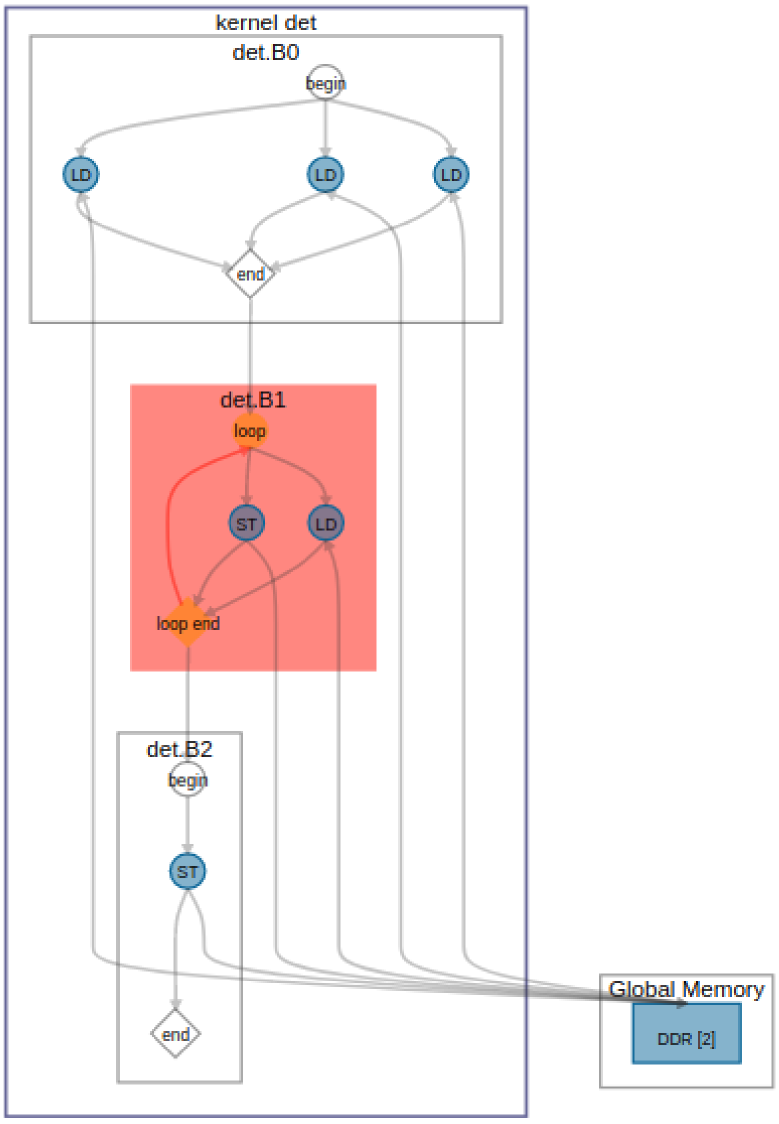


Figure 3.6: A flow diagram of the generated kernel with improved memory accesses. The area hi-lighted in red shows that the compiler was not able to pipeline this section. This corresponds to the area in which we preform the reads or writes to the table.

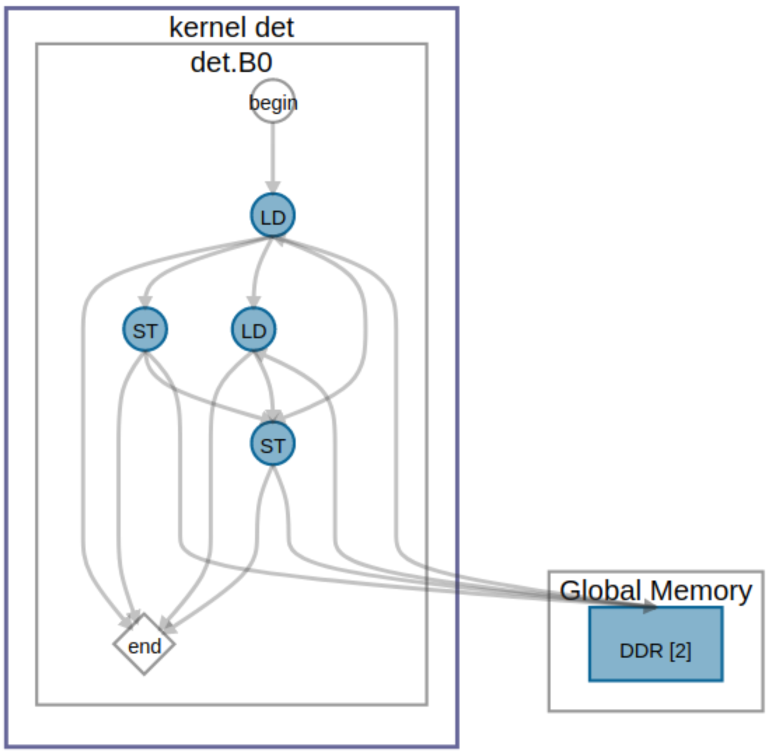


Figure 3.7: A flow diagram of the kernel with the improvements to memory accesses.

3.3.3 Deterministic CC on FPGA

The implementation for deterministic concurrency control on the FPGA is very similar to that of our earlier CPU experiments. The key difference instead of the scheduler transferring transactions to the worker threads after locking their tuples, we instead pass work to the FPGA. When the kernel running on the FPGA processes a transaction, it can assume that it has the necessary locks to execute it and so the FPGA simply execute the operations to the table in global memory that make up the transaction.

When running the implementation initially, we found that the application was spending the majority of the time filling the buffer of transactions to be executed, and so for a large amount of the benchmark the FPGA was idling. When inspecting the execution stack with the VTune performance analysis tool [16], it was noticed that half of the execution time was spent in the lock manager. In particular, operations on the list structures keeping track of the transactions waiting to acquire access to a tuple were dominating. Actions like adding to a list can be expensive due to memory allocations or traversing pointers however they are necessary as we need to keep track of dynamic amounts of transactions waiting on locks at runtime. This led to the FPGA idling for most of the applications runtime as seen in figure 3.8. If we continued with this lock manager, there would be little point in executing on the FPGA as it would simply be bottlenecked by the performance of the lock manager.

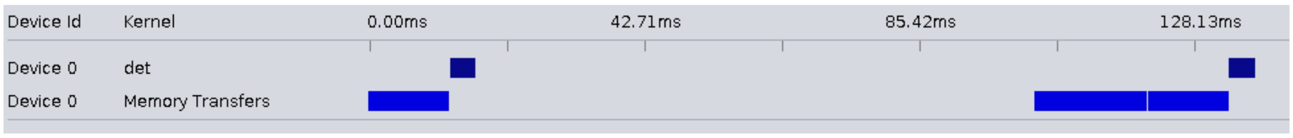


Figure 3.8: A timeline of the kernel execution with the initial lock manager from Intel's FPGA profiling tool. Note that there is a large amount of time between the two kernel executions where the FPGA is idling.

However, a key insight we had was that since we were now processing transactions in batches we do not to keep a track of all of the transactions that have a lock or are waiting to acquire one. This is because each batch is executed in isolation and all locks that are taken for one batch will be free on the next. Thus, much simpler data structures and logic can be employed to manage locking. The algorithm we use is as follows:

Algorithm 1 Batch locking algorithm

```
$readers \leftarrow \{0\}$
$writers \leftarrow \{0\}$
**while** *buffer* not full **do**
    $transaction \leftarrow$ pop front $transactions$
    $(type, key) \leftarrow transaction$
    **if** $type = $ **write then**
        $success \leftarrow \neg(readers(key) \vee writers(key))$
        **if** *success* **then**
            $writers(key) \leftarrow$ **true**
    **else**
        $success \leftarrow \neg(writers(key))$
        **if** *success* **then**
            $readers(key) \leftarrow$ **true**
    **if** *success* **then**
        push back *transaction* to *buffer*
    **else**
        push back *transaction* to *transactions*
```

Note that if a transaction contends with a transaction that has already been added to the buffer, we simply try that transaction again later. Since the *readers* and *writers* are now simple sets, we can implement them as simple bitmaps that have the length of the number of tuples in the table. Compacting these structures down helps to increase the locality of our accesses while locking and keep these sets entirely in cache. Also, the releasing of the locks done at the start can simply be implemented as an efficient `memset` operation of the sets to 0.

Figure 3.9 demonstrates the performance increase in locking batches of transactions with this new lock manager. We see a almost 4X speed-up when running YCSB B (read heavy) and almost of 5X speed-up in YCSB A (write heavy). When we run YCSB A we get effectively more contention when we use read-write locks with a higher number of writes. This then means that the lock manager has to try more transactions in order to fill the buffer for the batch. This leads to the performance decreasing for the original lock manager, as when a transaction cannot acquire it's locks we must keep a track the transaction's position involving map and list writes. Whereas in the improved lock manager, if a transaction cannot acquire it's locks we simply retry it later so we do not suffer any performance decreases from a higher proportion of writes.

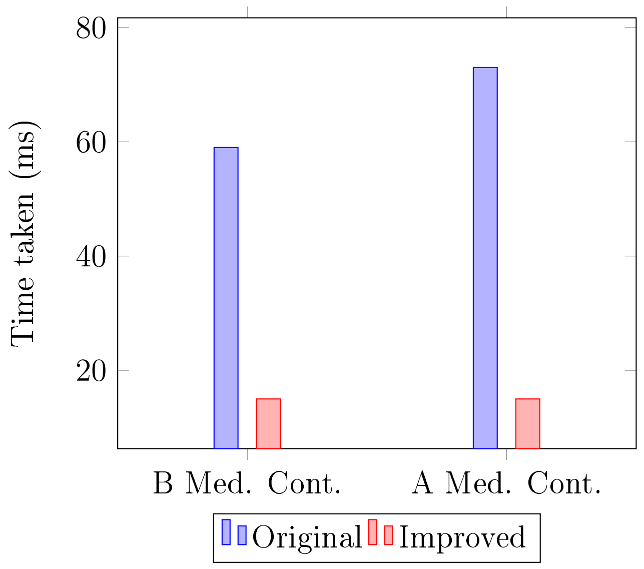


Figure 3.9: Average time taken locking a batch of 131072 transactions running YCSB A and B at medium contention (theta=0.5) with the original and improved lock managers.

The improvement can be seen in figure 3.10, where now the time that the FPGA is idling is significantly reduced. Overall because of the improved times to lock batches, we see an increase in overall throughput of 3.9 million transactions a second when running YCSB B with medium contention.

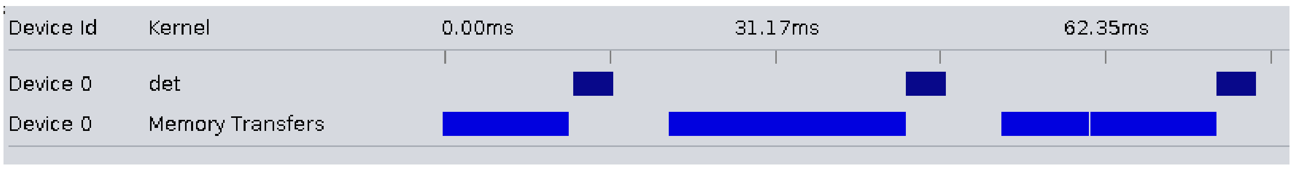


Figure 3.10: A timeline of the kernel execution with the improved lock manager from Intel's FPGA profiling tool.

3.3.4 TPL on FPGA

Two phase locking maps very easily in implementation to an FPGA. Since the workers manage concurrency control as well as the transactions, the only other part needed is a schedule to give transactions to the workers. This then is a natural fit to have the CPU host schedule and buffer the transactions to be sent over PCIe to the FPGA to be executed. The workers can then be simply implemented as compute units on the FPGA.

The main challenge of this design is to create a shared locking mechanism between the compute units, to allow the isolation of transactions that two phase locking gives us. In the CPU based implementation, we could easily achieve this through a shared table of read write locks. However, OpenCL does not natively support synchronisation between work groups. This is because OpenCL allow devices to schedule work groups to compute units during runtime, and so is abstracted away from the programming environment. However OpenCL does support a set of atomic functions in the standard, of which Intel supports the 32-bit atomic functions [14]. Of these functions, we have:

- `int atomic_cmpxchg (volatile __global int *p, int cmp, int val)` - Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute *(old == cmp) ? val : old* and store result at location pointed by *p*. The function returns *old*. [26]

- `int atomic_xchg ( volatile __global int *p, int val)` - Swaps the *old* value stored at location *p* with new value given by *val*. The function returns *old*. [26]

From this, we can then define our locking primitives as follows:

```
#define LOCK(a) while(atom_cmpxchg(a, 0, 1))
#define UNLOCK(a) atom_xchg(a, 0)
```

We also put mem fences after locking, and before unlocking to ensure that all accesses to global memory have finished after/before we lock/unlock. All together, this mechanism is an analogue of a spin lock in CUDA [1] common used in GPU programming to synchronise threads.

The final component that is needed to finish the locking mechanism is a table of locks in memory that can be shared between the compute units. OpenCL's memory model only allows *global* memory to be shared between compute units (*local* memory is shared within compute group, and *private* is shared within work groups). Ideally, we would have the lock table in the FPGAs internal memory to make use of it's high bandwidth. However, global memory is by default mapped to the FPGAs external memory and although the Intel compiler does support using heterogeneous memory for global memory, this is not a feature that our board supports. Therefore we implement this lock table as a buffer in global memory, and pass a pointer to it as an extra argument to the kernel. Additionally, on the initialisation of the kernel we zero out this buffer from the host to ensure all locks are available when the kernel starts.

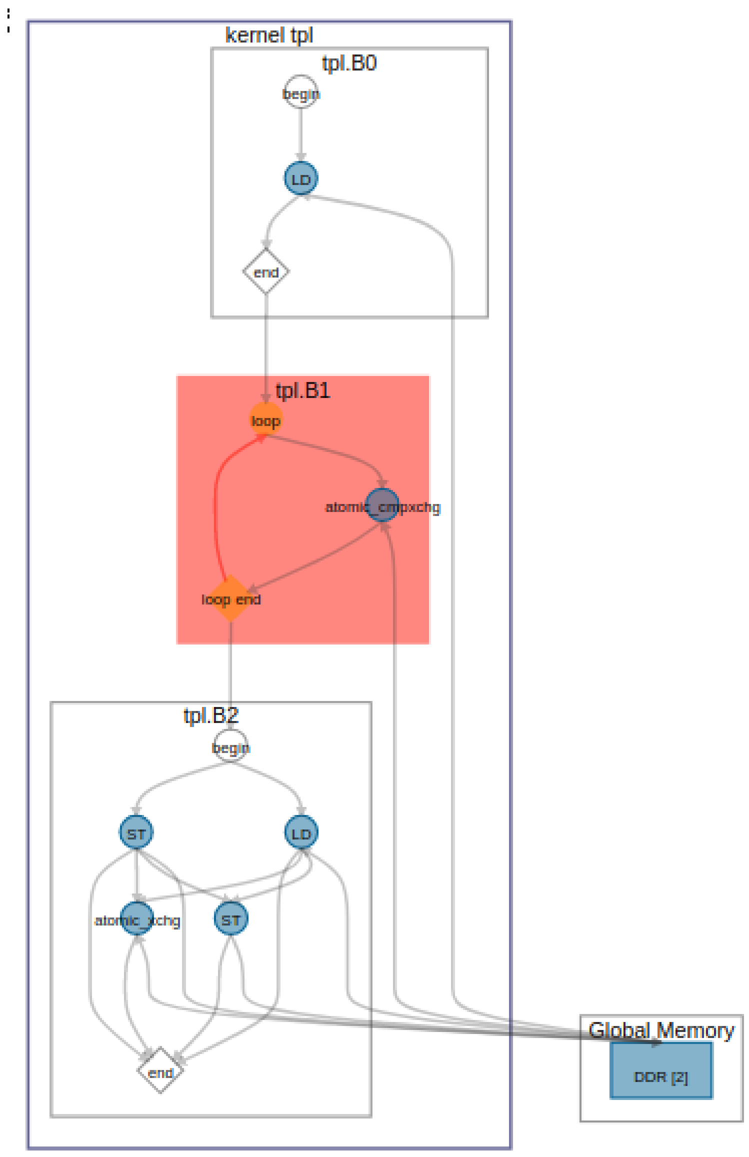


Figure 3.11: A flow diagram of the generated kernel for TPL.

Chapter 4

Evaluation

In the following section we will evaluate the performance of the deterministic and TPL concurrency control implementations on both the FPGA and CPU. In section 4.1 we will compare the performance of the two concurrency control schemes running purely on the CPU. Section 4.2 will then compare the concurrency control schemes running with the FPGA. Then in section 4.3 we will compare the absolute performance of all four implementations to assess the relative performance of each approach.

In our tests, we will use the YSCB benchmarks [5] to gauge performance. However, we have made slight modifications to the schema in order to generate more efficient hardware for the FPGA. Firstly, we reduce the size of fields from 100 bytes to 64 bytes. This is so that the records are aligned in memory, and that reads and writes will be a full cache line to ensure efficient LSUs on the FPGA. We also reduce the number of fields in each record from 10 down to 8, so that addresses of each field can be efficiently calculated on the FPGA. We found that these changes did not have a significant impact on the performance characteristics of the CPU implementation.

It should also be noted that we use a scrambled Zipfian distribution for the distribution of keys to spread the popular keys across the key space so that we avoid any caching that would not be seen in a real world workload. All read and write transactions are also single field.

Details about the hardware we run these experiments on can be found in the appendix in section A.1.

4.1 CPU baseline

In this section we will explore the performance characteristics of the two CPU based implementation to establish a baseline level of performance. For each benchmark we vary the number of workers executing transactions on each system. In addition to these threads, we also create a single thread for a scheduler to allocate work to the workers.

4.1.1 YCSB B

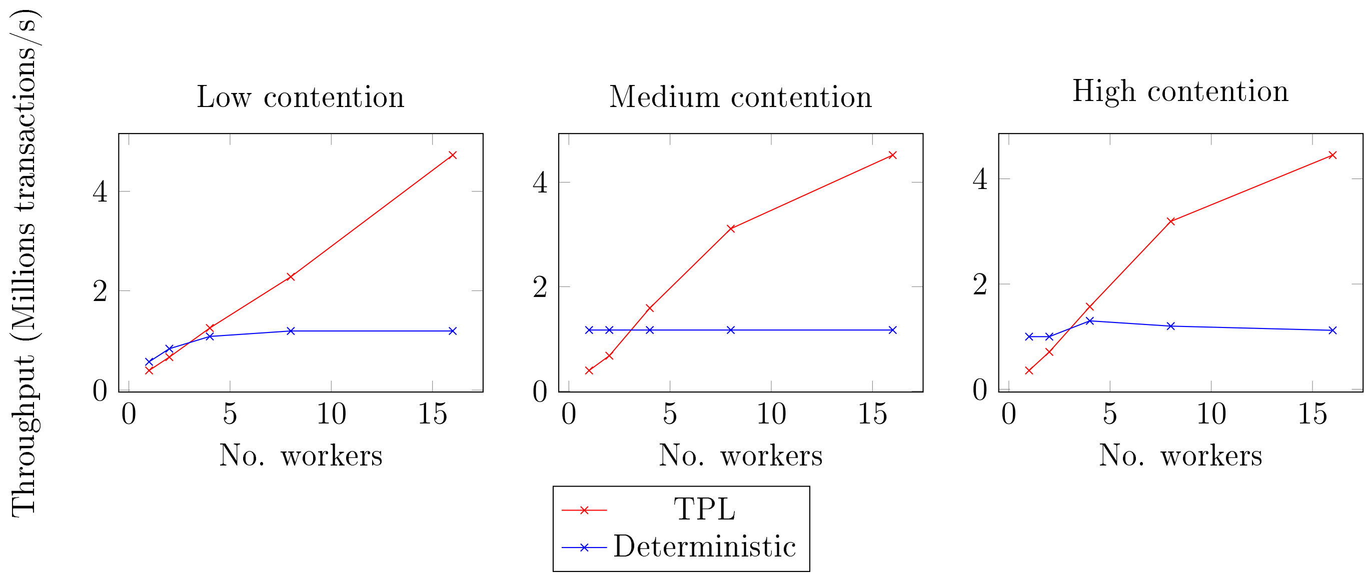


Figure 4.1: Throughput of transactions from YCSB B on the TPL and deterministic CPU based implementations with varying numbers of workers and contention levels (Low theta=0.1, Medium theta=0.5, High theta=0.9)

In these benchmarks we see that TPL far out scales the deterministic implementation by a large degree. Initially, we see that the deterministic implementation achieves a higher throughput than the TPL with small numbers of workers (1 or 2). This is because of the higher overhead of using CPU locks in the TPL implementation, vs in memory structure of the lock manger in the deterministic.

However once the number of workers increases beyond this, we see that the TPL implementation increases linearly with the number of workers while the deterministic remains flat. This is due to the fact that the deterministic implementation is reliant on the lock manager to schedule transaction. Since this lock manger is single threaded (to maintain determinism), the whole system is limited to the throughput of this stage. As we have discussed in section 3.3.3, the lock manager is slow due to the structures keeping track of waiting transactions. This is not a bottleneck in Calvin as it is a distributed system, and so the throughput of each node reaching in the tens of thousands [31]. However, in our single node setup this is a major issue for scalability.

The results do not significantly change over the varying levels of contention in this benchmark. This is largely due to the fact that this is a read heavy benchmark (95% reads, 5% writes) and both implementations use read-write locking per record. It can be noted however that the higher contention results do have a slightly faster scaling with the number of workers. This is due to the higher density of transactions on popular keys, leading to an increase in cache hits on records.

4.1.2 YCSB A

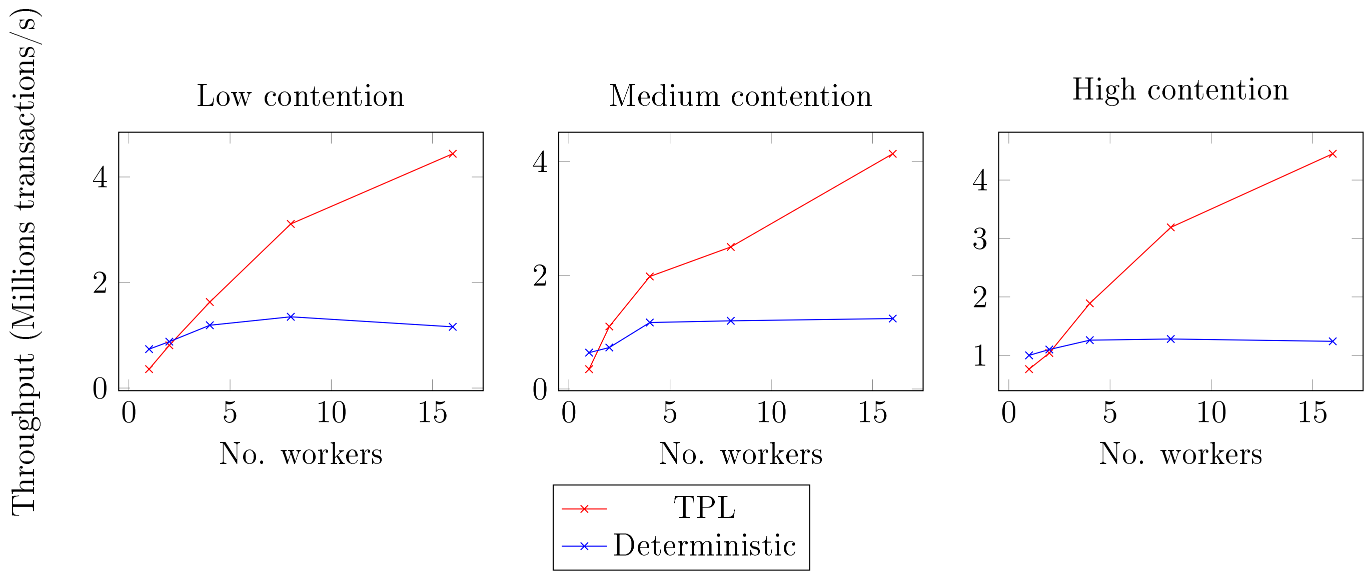


Figure 4.2: Throughput of transactions from YCSB A on the TPL and deterministic CPU based implementations with varying numbers of workers and contention levels (Low theta=0.1, Medium theta=0.5, High theta=0.9)

The results from YCSB A do not change signifigantly from the results for YCSB B despite the larger proportion of write transactions. Although there is a higher level on contention on our read-write locking mechanisms in this benchmark, since the transactions are single row and relatively short lived locking of the transactions continues not to be a bottleneck.

4.2 FPGA results

In this section we will evaluate the performance of both deterministic and TPL concurrency control running on the FPGA. We measure the FPGA's performance by compiling our designs with the flag `-profile=all` to the Altera offline compiler. This compiles additional hardware into the design to measure stalls, memory performance and other runtime characteristics of the kernels execution. We will use these results to compare the two designs, and identify bottlenecks in these techniques.

4.2.1 Kernel execution times

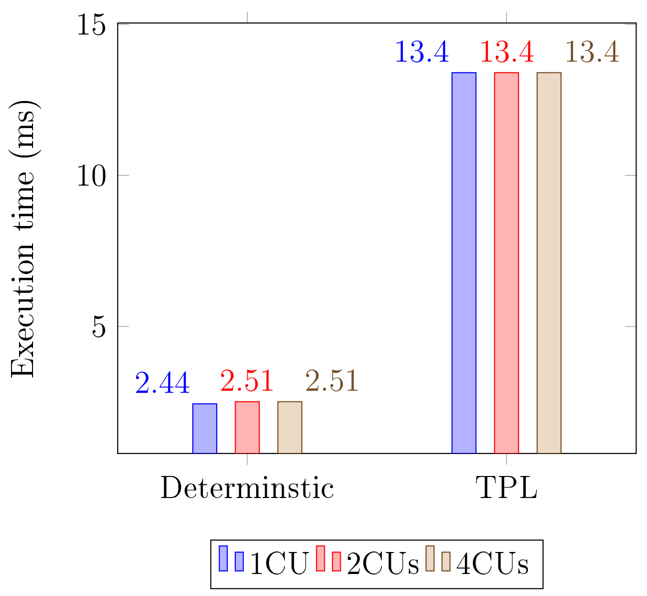


Figure 4.3: Kernel execution times of the deterministic and TPL designs running YCSB B with theta=0.1.

We can see from the kernel execution times that the deterministic design yields over a 5X speed-up over the TPL. Although the OpenCL report only shows 12 cycles being allocated to the lock acquisition out of a total of 213 for each work item in the pipeline, we see much worse performance than this when we run benchmarks. This is because each record has a lock that is only 32 bits wide, and since our access pattern will be very random (because of the scrambled Zipfian distribution) the LSUs will not be able to coalesce together lock acquisitions and releases in the pipeline leading to poor memory performance. This leads to 66% of lock acquisitions and 74% of lock releases stalling the pipeline, and a total utilisation of the memory bandwidth of 6.27% and 6.29% for these instructions respectively. In comparison, the deterministic kernel achieves 100% bandwidth efficiency on all memory instructions, and a 9% stall rate on the worst memory instruction.

We do however not see any scaling in either of the designs as more compute units are added. Although this should give more performance since we have multiple pipelines executing in parallel, the limiting factor is the available memory bandwidth and not the compute on the FPGA. We can see this in the profiler data from these runs. In the deterministic kernel with one compute unit, we achieve a memory bandwidth of 171.5MB/s and 3320MB/s for writes and reads to the table respectively. When we increase this to four compute units this decreases to 42.6MB/s and 824MB/s. This is almost exactly a 4X decrease in the memory bandwidth for every compute unit showing that any speed up we gain by the parallel execution of the pipelines is lost by a reduction in the individual performance of the pipelines.

No. Compute Units	Read speeds (MB/s)	Read stalls	Write speeds (MB/s)	Write stalls
TPL				
1	599	68.58%	35.5	40.75%
2	299	52.31%	15.7	8.46%
4	149.6	58.75%	7.9	48.11%
Deterministic CC				
1	3320	14.63%	171.5	2.67%
2	1654.9	9.39%	85.5	5.73%
4	823.7	9.25%	42.6	5.94%

Table 4.1: Memory performance of the deterministic and TPL kernels on reads and writes to the table with varying numbers of compute units. Performance is gathered from the kernel with the Intel profile tool while running YCSB B with theta=0.1.

4.2.2 Memory transfer times

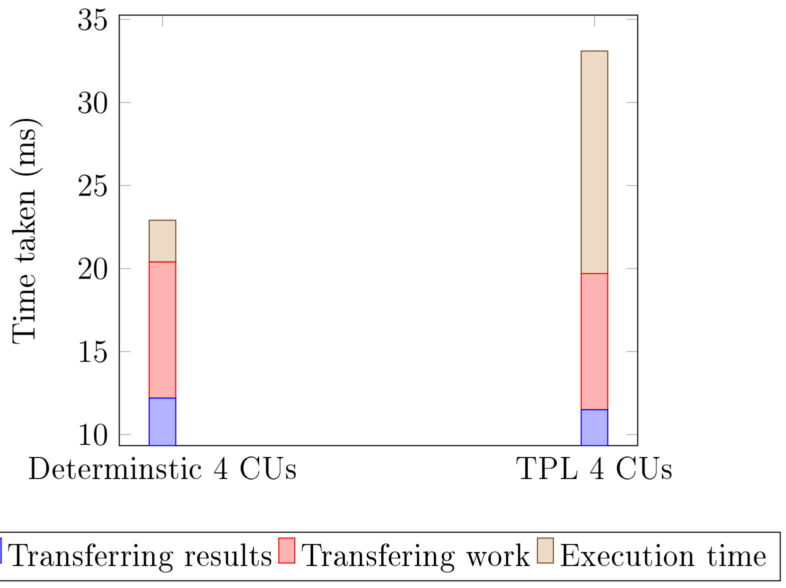


Figure 4.4: A breakdown of the time spent on the FPGA for both designs running YCSB B with theta=0.1. Shown is the time taken transferring work to the FPGA from the CPU, executing the transactions on the FPGA, and transferring the results from the FPGA to the CPU.

Between each run of the kernels, we must read the results from the FPGAs memory and write the new batch of transactions to be executed over the PCIe connection. Since the performance of this is mainly based on the memory bandwidth on the FPGA as well as the speed of the PCIe connection, these transfer times are constant for both designs. However, with the deterministic design's much faster kernel execution time, the time spent transferring memory dominates the time spent actually executing transactions. This therefore limits the possible performance of transaction processing on our FPGA card as increases in the performance of the FPGA design beyond this will have little impact on the overall throughput of the system.

4.2.3 FPGA resource utilization

No. Compute Units	ALUTs	FFs	RAMs	DSPs
Deterministic CC				
1	47593 (10%)	66496 (7%)	431 (17%)	0 (0%)
2	54480 (12%)	78526 (8%)	518 (20%)	0 (0%)
4	68252 (15%)	102605 (11%)	692 (27%)	0 (0%)
TPL				
1	47627 (10%)	66562 (7%)	431 (17%)	0 (0%)
2	54548 (12%)	78658 (8%)	518 (20%)	0 (0%)
4	68324 (15%)	114865 (12%)	700 (27%)	0 (0%)

Table 4.2: The FPGA resource consumption of the deterministic CC and TPL kernels with varying number of compute units. Percentages in the columns show the proportion of the boards total available resources the design consumes.

As demonstrated in table 4.2 both the Deterministic and TPL kernels take up a small amount of our FPGA's available resources, even when scaling up the number of compute units. This is due to the both kernels mainly being focused on accesses to global memory, and so most of the resources used are for the generation of LSUs. Thus more compute intensive resources such as DSPs are not even utilised in these designs.

We also see that the scaling up resources for both designs is fairly linear with the number of compute units added. Since adding more compute units in OpenCL simply adds duplicate pipelines, and communication or synchronisation is not needed between them (accept for atomic functions used in TPL) this behaviour is to be expected.

Typically in most FPGA applications, you come up with an optimised design for the task and find a way to scale it up to maximise the usage of the available hardware on the FPGA to further increase performance. However, since in our application we have seen that we are extremely memory bound, we do not benefit from using the additional hardware to create multiple compute units. This is not necessarily a drawback of transaction processing on an FPGA as additional area could be used to implement more complex transaction types, offload other tasks such as query processing to the FPGA, or simply allow us to run this application on smaller and more cost effective FPGAs.

4.3 CPU vs FPGA performance

4.3.1 Throughput

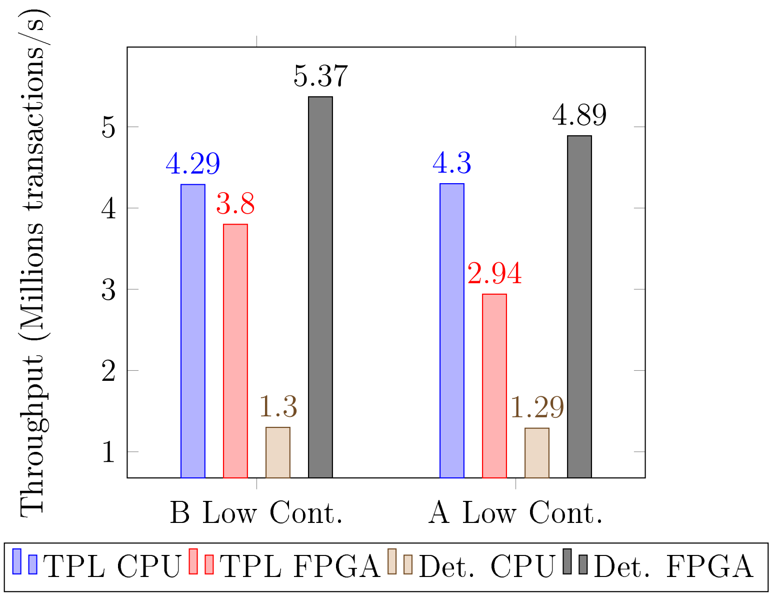


Figure 4.5: Overall throughput of TPL and deterministic implementations running on the FPGA and the CPU for YCSB A and B with theta=0.1

While we have seen that the kernel execution time of the TPL on the FPGA gives it a higher throughput then we get from TPL running on the CPU, the overall throughput remains slower for both benchmarks. This is due to the large amount of overhead that occurs from transferring work and results to and from the FPGA.

We also see a large disparity between the CPU and FPGA results in the deterministic systems. This is thanks to the much faster batch lock manager we use for the FPGA. Since this lock manager does not incur any overhead as we have seen, we are able to run at the full speed of the FPGA, which we have seen has an extremely high throughput.

Overall, we see that the deterministic FPGA implementation has the highest overall throughput. However, it should be noted that this lead does somewhat decrease in the YCSB A benchmark with a higher proportion of writes. This is due to the fact that writes to DRAM are slower than reads. Since we have seen that the FPGA performance is highly memory bound, this has a large impact on its performance. Whereas, on CPU implementations are not memory bound and so do not show any fluctuation in performance between the two benchmarks.

4.3.2 Latency

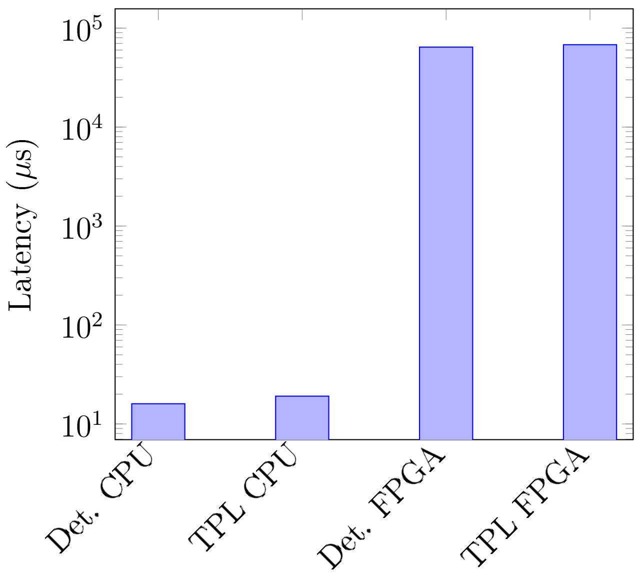


Figure 4.6: The latency of a single read transaction from start to finish with TPL and Deterministic CC both on the FPGA and CPU implementations. Note that we include the time taken to fill the entire buffer of transactions in the batch in the FPGA experiments.

Although the FPGA implementations do achieve a high throughput (as demonstrated in the previous section), as seen in figure 4.6, the trade-off we make to achieve this is increased latency of individual transactions. In particular, we see around a 4000X increase in latency for our FPGA implementations vs the CPU based equivalents. This increase is a product of moving to batch processing as well as the additional latency of transferring data over the PCIe bus. With an overall latency of around 60ms for the FPGA implementations this should still be suitable for most applications, however for applications that require low latencies for individual transactions we see that a CPU based implementation will be preferable.

While the CPU implementations have significantly lower latencies than both FPGA implementations, it can be noted that the deterministic CC CPU implementation has a lower latency of 16μs compared to the 19.1μs of the TPL. From observing their execution in VTune, we find that this is due to the locking in the deterministic lock manager faster than that of the CPU locks used in the TPL implementation. It is also worth noting that the Deterministic FPGA implementation also outperforms its TPL counterpart in latency by around 3.7ms due to its shorter overall kernel execution time as seen in figure 4.3.

It should be noted that although the latency of both FPGA implementations theoretically could be much lower with future hardware improvements. The pipeline of the deterministic kernel takes 189 cycles to complete, whilst the TPL implementation takes 195. If both kernels were able to achieve the full clock speed of 500MHz and experienced no stalls on memory, the latency of a transaction would be 0.37μs and 0.39μs respectively. Thus, it is possible that we could end up with latencies smaller than the CPU implementations with future improvements to interconnect between CPU and FPGA.

Chapter 5

Conclusion

Using deterministic concurrency control methods is an exciting advancement in the field of distributed OLTP databases. This simpler method of concurrency control enables far better scaling across large numbers of nodes, by bypassing much of the synchronisation needed to maintain ACID properties.

For a single node database system however, traditional concurrency schemes still remain on top. We have shown through our CPU experiments that TPL based concurrency control far out-scales deterministic. Because of the low cost of synchronisation of thread in a single node compared to over multiple nodes, the single threaded lock acquisition of deterministic concurrency control becomes a serious bottleneck to its scaling.

However, when processing transactions on an FPGA, the bottlenecks and limiting factors change significantly. As demonstrated, the cost of managing concurrency through locking values on an FPGA is extremely high because of the resulting small and inefficient atomic actions on global memory. This then results in a significant drop in performance when executing TPL on the FPGA as locking for each transaction add a high amount of stalling to the pipeline, reducing parallel execution. Whereas, deterministic transaction processing maps well to an FPGA's architecture as the design of the kernel can be simple and pipelined. Additionally, where the lock manager was a bottleneck to performance on the CPU implementation we have put forward a new lock manager optimised for batch processing which is necessary for the FPGA. This yields much higher throughput in the lock manger, but also allows processing of the locking in parallel to the kernel execution on the FPGA.

While TPL remains slower on the FPGA than on the CPU, deterministic transaction processing on the FPGA shows a notable increase in overall throughput over anything that we saw was possible without it. This is thanks to processing the transactions in a very efficient pipelined design, that is able to fully saturate the memory bandwidth available to it.

Overall, the application of FPGAs to in memory transaction processing is very promising. As technology improves, we expect the case for transaction processing on FPGAs to get stronger. During this projects experiments, it was found that transaction processing on the FPGA is highly memory bound. From this we anticipate that increases in memory bandwidth available to the FPGA would result in equal increases in throughput of transactions. Manufacturers such as Intel and Xilinx have already started to create FPGA devices with *high bandwidth memory* (HBM) [15] [35] that are capable of memory bandwidth of around 1 TB/s compared to the 6.4GB/s bandwidth of the DDR3 memory on our FPGA. Additionally from the experiments it was found that a large amount of overall execution time is spent transferring data back and forth between the device. This bottleneck will only get smaller with improvements in the interconnect between the CPU and FPGA.

5.1 Challenges

Many challenges were encountered during the course of this project. Although steps were taken to mitigate against some of these challenges, there was still a number of hurdles to face along the way.

Setting up the FPGA programming environment proved difficult at first. Although Terasic provides a board support package for the FPGA board, the installation process was not straightforward. The server used for this project was running Ubuntu as it's Linux distribution, however Intel only officially supports CentOS for Linux. This resulted in having to modify the drivers for the board in order for them to compile and be installed. Additionally, the default Altera driver as this was causing issues with the board not being recognised by the system and had to be manually disabled. Lastly, physical access the system was required to attach a cable to the boards JTAG interface to program the board with an OpenCL kernel so that the device could be further reprogrammed through the PCIe connection.

Although OpenCL was chosen in this project for ease of programming over HDL languages like VHDL or Verilog, issues were still experienced with this environment over traditional CPU programming. Firstly, although OpenCL abstracts the hardware generation away from the programmer, it still has to synthesize the hardware and its placement - the same as compiling a HDL program. This causes large compile times for even simple designs, and so typically design iterations took around 3 to 4 hours. This limited the pace at which new designs could be developed and tested on the FPGA during the project.

Additionally, the debugging tools available for the OpenCL kernels did present challenges. Unlike a traditional program where a debugger can be used to stop and inspect the program state at a given point in time, it is not possible to do this with a kernel running on an FPGA. Intel does provide an option to compile the kernel for emulation which does allow it be to ran in GDB and inspect state. However, when it was used it was usually not adequate as the emulator does not accurately reflect the hardware of the FPGA making issues that we were seeing on the FPGA not appear on the emulator. Printing is also supported from the FPGA from Intel's OpenCL compiler, which does allow peeping into the execution state of the FPGA. However, in practice this is not too useful as a tool as prints only occur when kernel execution is finished so kernels that stall cannot be observed. Additionally, `printf` statements take up a large amount of resources, limiting the number of complexity that can be used. It was found that the best way to overcome this was to read and verify the global buffers on the host after kernel execution, however this meant that to check any internal state we had to compile in writes to these buffers.

5.2 Future work

While this project has provided a basis for transaction processing on FPGAs, there are many areas that could be explored further that were beyond the scope of this project.

5.2.1 Optimistic concurrency control

This project primarily experimented with with deterministic concurrency control, and TPL as different methods of concurrency control for transaction processing on FPGAs. From the results it was found that deterministic concurrency control is indeed a good fit for the task. However, there are still more types of currency control that could be explored for transaction processing on FPGAs.

In particular, an interesting form of concurrency control to investigate would be *optimistic concurrency control*. Optimistic concurrency control (OCC) was first introduced in 1981 as a way to avoid the overhead of lock maintenance and deadlock [20]. OCC adds an additional *verification* step to the end of every transaction. This step verifies that during the transaction, no other transaction has written to the data it has read. In the case of a conflict, the transaction is then rolled back. This then removes any need for locking during executing transactions. Silo [33], a leading in memory single node system, uses a form of optimistic concurrency control.

This could potentially be a good fit for transaction processing on an FPGA, because of it's lack of locking during executing transactions. As the findings of this project demonstrates, locking on an FPGA is a major bottleneck when implementing TPL. The key to this investigation would be to see if compared deterministic concurrency control (also lock free) is executing the verification step on the FPGA advantageous compared to the batch locking on the CPU.

5.2.2 Improving utilization of FPGA resources

One characteristic of the FPGA implementations found in the evaluation is that the OpenCL kernels take up little of an FPGA's total resources. This is because most of the performance is achieved through efficient memory accesses and not computational speed. However, this is a large waste of the FPGA's resources that could be used to otherwise speed up the database. Therefore a natural extension of the project would be to explore ways to use this additional area.

One possible use could be to implement a caching system for the table. We hypothesise that since our current kernel's performance is bound by the speed of memory accesses, better caching would lead to higher memory performance and thus improve the overall throughput of the kernel. Even with 4 compute units in the current deterministic kernel, only 27% of the on-board memory leaving approximately 36.5 Mbit of on board memory capacity for a cache. Although small in capacity, these M20K memory blocks have a significantly higher throughput and lower latency than accessing the DDR memory off the chip. Additionally, these memories can be clocked at double the frequency of the OpenCL kernels allowing the memory to be *double pumped* resulting in twice the bandwidth [13]. Thus for workloads with small hotspots, using more resources for a cache could significantly increase the throughput of the kernel.

Alternatively, the extra area could be used to offload further work from the DBMS to the FPGA. Query processing, proven to be power on an FPGA by Ibex [36], could be done in tandem with the transaction processing on the FPGA and could utilise the FPGAs internal interconnect for extremely high bandwidth data transfer between them. Not only would this accelerate these individual tasks, but could also reduce the overall data being transferred between the CPU and FPGA by performing selections and projections on the FPGA. As shown in section 4.2.2, memory transfer times dominate kernel execution time so reducing this could be a major benefit.

5.2.3 Non-volatile storage

During this project, the focus was on in-memory databases. However, the limited capacity of random access memory is often too constraining for many real world applications. This would further be exacerbated by the move to HBM as suggested earlier as the current HBM2 specification only allows up to 8GB per stack. Thus, a natural further extension is to investigate support for non-volatile storage.

Connecting storage such as an SSD or HDD to the FPGA board could easily be done through the board's four SATA ports that connect to the Stratix V device. Additionally, Intel makes available a SATA Ip core which provides link layer to implement SATA channel to the FPGA [17].

The disadvantage of using deterministic concurrency control with non-volatile storage is that stalls on the storage can be costly to performance, since transactions must be executed in a deterministic ordering. Calvin solves this problem by delaying any transaction that may incur a disk stall before it is scheduled while it sends out a request to retrieve the relevant records [31]. However, this solution may not map well to the architecture set out in this project as communication between the scheduler and the storage would have to take place over the PCIe connection which would most likely incur a larger delay than would be worth to reduce stall times. This leaves an open problem to solve, and perhaps a novel new architecture or algorithm could be employed to solve it.

Appendix A

Main Appendix

A.1 Hardware

A.1.1 Server

All of our experiments were run on the Taz research server managed by the LSDS group. This server features two Intel Xeon Silver 4114 Skylake CPUs running at 2.2GHz. Each of these CPUs has 10 physical cores and 20 logical cores, and a 13.75MB L3 cache. The CPUs are connected via a NUMA interface. Both CPUs feature 6 memory channels, which are each populated with 16GiB DDR4 DIMMs running at 2666MHz.

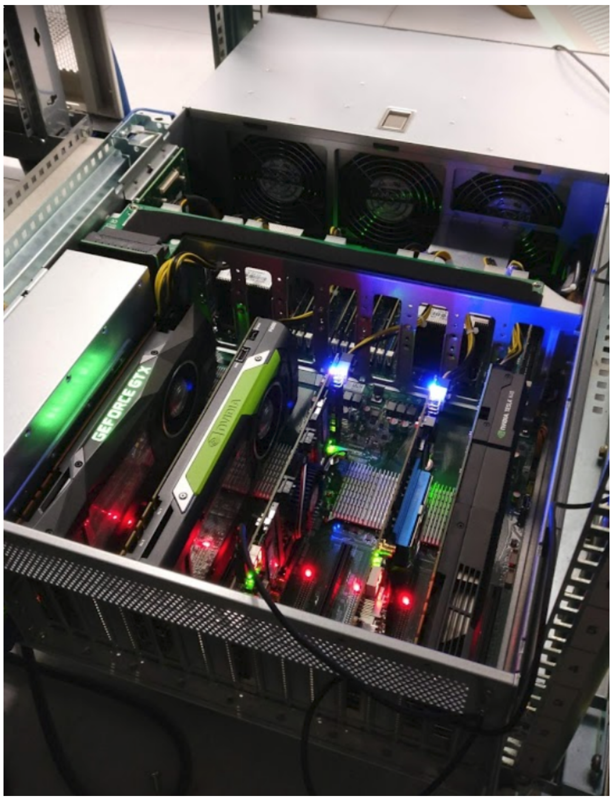
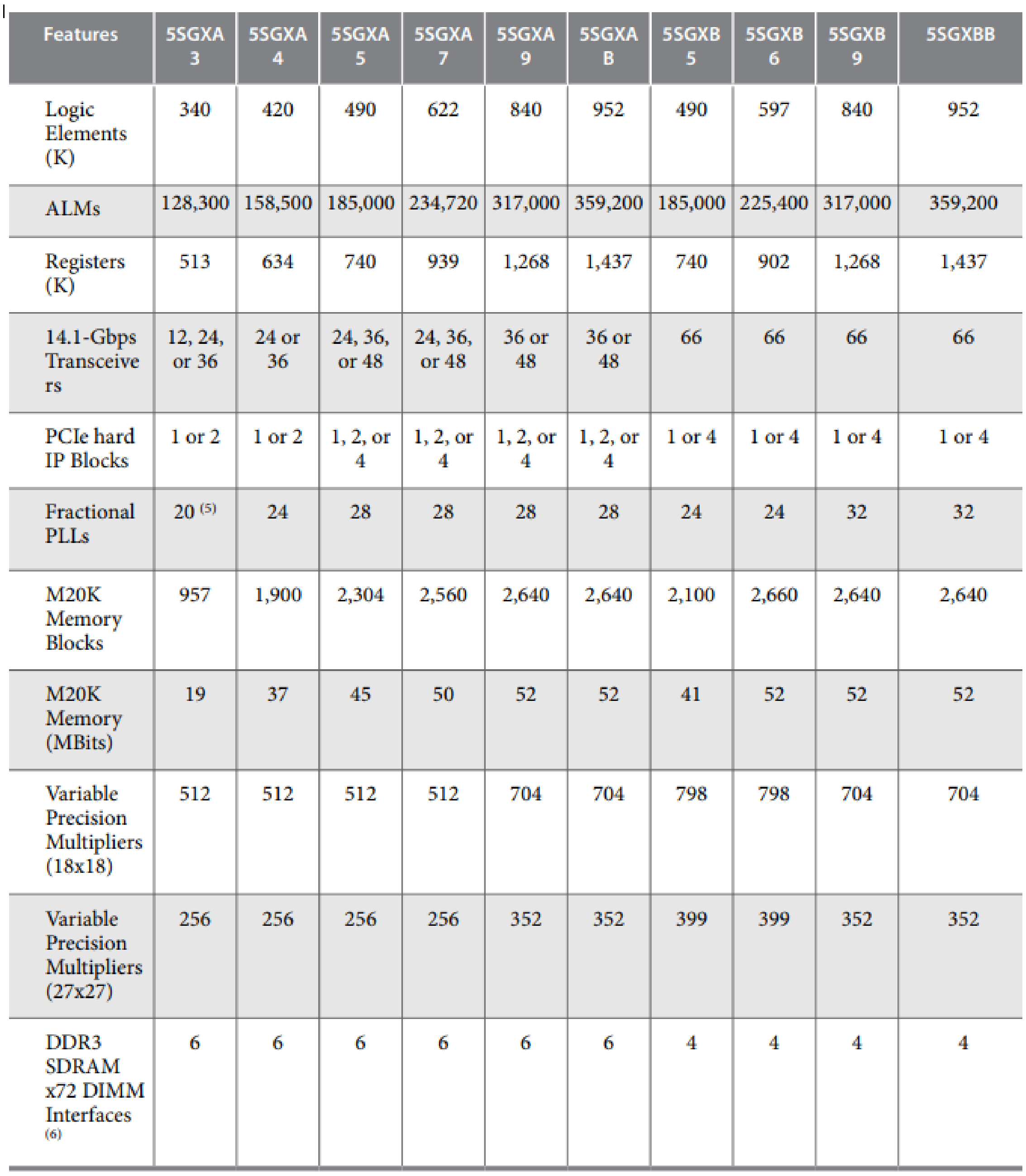


Figure A.1: The Taz research server. The DE5-Net card can be seen as the third card from the left.

A.1.2 FPGA board

For the FPGA implementations, we used the DE5-Net FPGA Development Kit from Terasic. This development board features the Altera Stratix V GX FPGA (5SGXEA7N2F45C2), which features a large amount of hardware resources as seen in the 5SGXA7 column of figure A.2 and is capable of running at a maximum speed of 500MHz. For the main memory, the board is configured with 2, 2GB DDR3 DIMMs. The board is connected to the server via a x8 PCIe Gen 3.0 connection, capable of a maximum total throughput of 8GB/s between the device and host.



Features	5SGXA 3	5SGXA 4	5SGXA 5	5SGXA 7	5SGXA 9	5SGXA B	5SGXB 5	5SGXB 6	5SGXB 9	5SGXBB
Logic Elements (K)	340	420	490	622	840	952	490	597	840	952
ALMs	128,300	158,500	185,000	234,720	317,000	359,200	185,000	225,400	317,000	359,200
Registers (K)	513	634	740	939	1,268	1,437	740	902	1,268	1,437
14.1-Gbps Transceivers	12, 24, or 36	24 or 36	24, 36, or 48	24, 36, or 48	36 or 48	36 or 48	66	66	66	66
PCIe hard IP Blocks	1 or 2	1 or 2	1, 2, or 4	1, 2, or 4	1, 2, or 4	1, 2, or 4	1 or 4	1 or 4	1 or 4	1 or 4
Fractional PLLs	20 (5)	24	28	28	28	28	24	24	32	32
M20K Memory Blocks	957	1,900	2,304	2,560	2,640	2,640	2,100	2,660	2,640	2,640
M20K Memory (MBits)	19	37	45	50	52	52	41	52	52	52
Variable Precision Multipliers (18x18)	512	512	512	512	704	704	798	798	704	704
Variable Precision Multipliers (27x27)	256	256	256	256	352	352	399	399	352	352
DDR3 SDRAM x72 DIMM Interfaces (6)	6	6	6	6	6	6	4	4	4	4

Figure A.2: The Stratix V GX family features [18]

A.2 ACID

ACID (Atomicity, consistency, isolation, durability) is a set of properties that most modern databases guarantee. The definition of these properties, first published in 1983 by Haerder and Reuter [10], are as follows:

- **Atomicity** - All-or-nothing - either all of the actions in a transaction must be commited or none.
- **Consistency** - Any transaction that has been commited perseveres the consistency of the database. Eg, each commited transaction must bring the database from one valid state to another.
- **Isolation** - The concurrent execution of transactions must be equivalent to some serial ordering of the transactions being executed.
- **Durability** - Once a transaction has committed it's results, the results must survive any subsequent malfunction of the system.

A.3 Performance comparisons

We also set out to compare the performance achieved with our implementations with other similar leading in-memory OLTP databases. In particular, databases that implemented YCSB for a direct comparison. From this, we chose three databases: Silo [33], Deneva [38] and Cicada [23]. We ran these systems on our hardware in order to compare the results we achieved in this project to the state of the art.

However we found that after running these databases on our hardware, they produced wildly different results to our implementations as well as to each-other. Deneva managed a throughput of around 100,000 transactions a second, whilst Cicada achieved around 4 million transactions a second and around 10 million for Silo. This appeared to be due to the difference in implementations of YCSB between these systems as well as our inability to get some of the dependencies working. Overall, we decided to leave these comparisons out due to their inconsistencies and leave this to future work.

A.4 Source code

This section will show some of the more important source code. The full source code used in this project is available at https://gitlab.doc.ic.ac.uk/wjw15/hobbes.

A.4.1 OpenCL kernels

Deterministic CC

Listing A.1: The final kernel used for TPL

```
#include "../../backend/txn.h"

__attribute__((reqd_work_group_size(WORK_ITEM_SIZE,1,1)))
__attribute__((max_work_group_size(WORK_ITEM_SIZE)))
__attribute__((num_compute_units(NO_CUS)))
__kernel void det(__global Txn * restrict txns,
      __global Result * restrict results,
      __global Col * restrict table) {
   size_t gid = get_group_id(0);
   size_t lid = get_local_id(0);
   size_t i = (gid * WORK_ITEM_SIZE) + lid;
   __private Txn txn;
   __private Result res;
   __private size_t col;
   __private int key;
   __private size_t pos;

   txn = txns[i];
   key = txn.key;
   col = txn.col;
   pos = (key * 8) + col;

   mem_fence(CLK_GLOBAL_MEM_FENCE);

   if (txn.type == Write) {
        table[pos] = txn.value;
   } else {
        res.result = table[pos];
   }

   mem_fence(CLK_GLOBAL_MEM_FENCE);

   res.id = txn.id;
   res.success = true;
   results[i] = res;
}
```

TPL

Listing A.2: The final kernel used for deterministic CC

```
#include "../../backend/txn.h"

#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable

#define LOCK(a) while(atomic_cmpxchg(a, 0, 1) != 0)
#define UNLOCK(a) atomic_xchg(a, 0)

__attribute__((reqd_work_group_size(WORK_ITEM_SIZE,1,1)))
__attribute__((max_work_group_size(WORK_ITEM_SIZE)))
__attribute__((num_compute_units(NO_CUS)))
__kernel void tpl(__global Txn * restrict txns,
      __global Result * restrict results,
      __global volatile unsigned int * restrict lockTable,
      __global Col * restrict table) {
   size_t gid = get_group_id(0);
   size_t lid = get_local_id(0);
   size_t i = (gid * WORK_ITEM_SIZE) + lid;
   __private Txn txn;
   __private Result res;
   __private size_t col;
   __private int key;
   __private size_t pos;

   txn = txns[i];
   key = txn.key;
   col = txn.col;
   pos = (key * 8) + col;

   LOCK(&lockTable[key])
   mem_fence(CLK_GLOBAL_MEM_FENCE);

   if (txn.type == Write) {
        table[pos] = txn.value;
   } else {
        res.result = table[pos];
   }

   mem_fence(CLK_GLOBAL_MEM_FENCE);

   UNLOCK(&lockTable[key]);

   res.id = txn.id;
   res.success = true;
   results[i] = res;
}
```

Bibliography

[1] Alglave, J., Batty, M., Donaldson, A. F., Gopalakrishnan, G., Ketema, J., Poetzl, D., Sorensen, T., and Wickerson, J. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15, ACM, pp. 577–591. event-place: Istanbul, Turkey.

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

[3] Blott, M., Karras, K., Liu, L., Vissers, K., Bär, J., and István, Z. Achieving 10gbps line-rate key-value stores with fpgas. In *Presented as part of the 5th {USENIX} Workshop on Hot Topics in Cloud Computing* (2013).

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[5] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (2010), ACM Press, p. 143.

[6] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 205–220.

[7] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notions of consistency and predicate locks in a database system. *Communications of the ACM 19*, 11 (1976), 624–633.

[8] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., and Chung, E. Azure accelerated networking: SmartNICs in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 51–66.

[9] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management*

of Data (1994), SIGMOD '94, ACM, pp. 243–252. event-place: Minneapolis, Minnesota, USA.

[10] Haerder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (1983), 287–317.

[11] Harding, R., Van Aken, D., Pavlo, A., and Stonebraker, M. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment 10*, 5 (2017), 553–564.

[12] He, B., and Yu, J. X. High-throughput transaction executions on graphics processors. *Proceedings of the VLDB Endowment 4*, 5 (2011), 314–325.

[13] Intel. Intel FPGA SDK for OpenCL pro edition best practices guide. 191. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf.

[14] Intel. Intel FPGA SDK for OpenCL pro edition programming guide. 212. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[15] Intel. Intel stratix 10 MX FPGAs. https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html.

[16] Intel. Intel VTune amplifier. https://software.intel.com/en-us/vtune.

[17] Intel. SATA IP core. https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/designgateway-co---ltd-/ip/sata-ip-core.html.

[18] Intel. Stratix v device overview. 23. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf.

[19] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P., Madden, S., Stonebraker, M., and Zhang, Y. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1496–1499.

[20] Kung, H.-T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS) 6*, 2 (1981), 213–226.

[21] Kuon, I., Tessier, R., and Rose, J. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation 2*, 2 (2008), 135–253.

[22] Lamport, L. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

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

[24] Linden, G. Geeking with greg: Marissa mayer at web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[25] MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS) 11*, 4 (1986), 378–396.

[26] MUNSHI, A. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)* (2009), IEEE, pp. 1–314.

[27] SIROWY, S., AND FORIN, A. Wheres the beef? why FPGAs are so fast. *Microsoft Research, Microsoft Corp., Redmond, WA 98052* (2008).

[28] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal 30*, 3 (2005), 202–210.

[29] THE KRONOS GROUP. OpenCL - the open standard for parallel programming of heterogeneous systems, 2013. https://www.khronos.org/opencl/.

[30] THOMSON, A., AND ABADI, D. J. The case for determinism in database systems. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 70–80.

[31] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 1–12.

[32] TPC. TPC - current specifications. http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp.

[33] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 18–32.

[34] WALDROP, M. M. The chips are down for Moores law. *Nature News 530*, 7589 (2016), 144.

[35] WISSOLIK, M., ZACHER, D., TORZA, A., AND DAY, B. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. 11.

[36] WOODS, L., ISTVÁN, Z., AND ALONSO, G. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment 7*, 11 (2014), 963–974.

[37] YALEDB. Calvin source code. https://github.com/yaledb/calvin.

[38] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment 8*, 3 (2014), 209–220.