Investigating OS/DB co-design with SharedDB and Barrelfish

Master Thesis

Author(s):
Chothia, Zaheer

Publication date:
2013

Permanent link:
https://doi.org/10.3929/ethz-a-009935765

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 84

Systems Group, Department of Computer Science, ETH Zurich

Investigating OS/DB co-design with SharedDB and Barrelfish

by

Zaheer Chothia

Supervised by

Prof. Dr. Timothy Roscoe
Dr. Kornilios Kourtis
Jana Gičeva

December 2012–June 2013

inf | Informatik
Computer Science

Abstract

Databases and operating systems both operate over the same hardware but they have differing views of how the resources should be managed. As a result databases forgo the system services and implement their own scheduling and memory management which have been refined over several decades. Co-design is an area of research to address this by improving the collaboration between both systems. The idea is to integrate application knowledge in the operating system's decisions, and for the database to receive notifications and adapt to changes in system state.

This thesis presents work in this area which builds upon two research systems. One is Barrelfish OS, a realisation of the multikernel model which treats the machine as a distributed system. The other is SharedDB, an in-memory relational database designed around sharing of computation which delivers predictable performance on large and complex workloads. Our experimental results on Linux show that less than a quarter of the cores are needed to achieve peak performance when running a TPC-W workload. We are interested in resource consolidation and present an approach for utilising application knowledge in spatial and temporal scheduling of operators. We ported SharedDB to run on top of Barrelfish and conclude with discussion that the combined system makes a good foundation for testing new ideas and optimisations which bisect several layers of the system stack.

Contents

Chapter 1

Introduction

1.1 Context

In our day-to-day interactions we encounter numerous systems which store and transform vast quantities of data. Such applications have demanding requirements and need to support many end users with predictable performance. Databases are therefore especially concious of their resource usage. The problem is that some operating system policies are rigid and unsuited to the applications' needs [34]. Consequently, databases make limited use of the system services and instead allocate and manage resources on their own. For instance, they request large pools of memory from the system and sub-divide this internally. Another example is that they know whether I/O access is sequential or random and can implement custom buffer pool strategies and pre-fetching logic.

Database / operating system co-design is an area of research which aims to address some of these needs by exploiting application knowledge and improving the collaboration between both systems. One such project, Infokernel [4], provides visibility into the kernel's state and algorithms. In this way, default policies are transformed to mechanisms which can be controlled from user space. Another project, Cod [15], demonstrates the value of interaction between a database storage engine and an operating system's policy engine. The OS can incorporate database cost models when reasoning about scheduling and placement decisions. The database also receives notifications on changes to the system state and can adapt to meet SLA guarantees.

1.2 Problem statement

In this thesis we build upon two large systems and investigate synergies of combining them in a single scenario. On one side we have an operating system (Barrelfish [5]) which treats the machine as a distributed system; on the other is a database (SharedDB [14]) intended for predictability which leverages shared computation. Both projects are being developed within the Systems Group at ETH Zürich and in collaboration with external partners (Microsoft Research and Amadeus respectively). They are part of a larger goal of rethinking the application stack to better cope with increasingly challenging application requirements and modern innovations in hardware (SWissBox [1]).

The two systems make a natural fit because they both share the common design principle of minimising shared state and avoiding synchronisation allowing them to fully exploit multicore hardware. SharedDB presents an interesting use case to port on top of Barrelfish because it is a large application which spans the system. Furthermore, it has demanding performance requirements which stress several subsystems, from memory management through networking.

The main achievement of of this thesis is to make Barrelfish and SharedDB work together. Whilst there were a number of challenges along the way, this now opens new possibilities for future research. Beyond this, we conducted some experiments to understand the performance characteristics of SharedDB. These results demonstrate resource consolidation which still meets the application's SLA. We present some early work in this direction and discuss several avenues for future investigation.

1.3 Outline

Chapter 3 goes into more detail of the porting process and some of the challenges we faced. In Chapter 4 we present experimental results on specific aspects of SharedDB's performance. Chapter 5 returns to the topic of co-design and presents two interesting research questions. Finally Chapter 6 concludes with a discussion of what we have learned and provides outlook for further research.

Chapter 2

Background

In this section we introduce the key ideas and overall design of the systems being used throughout this thesis. Barrelfish incorporates ideas from distributed computing and applies these to operating system design. SharedDB is a relational database which processes queries in batches and is built around the concept of shared computation.

2.1 Barrelfish

Barrelfish is a research operating system designed to scale to modern multicore architectures. It is a realisation of the multi-kernel model [5] which treats the machine as a distributed system by placing an independent node on each core and replicating state by explicitly passing messages rather than sharing memory. This is based on the observation that the underlying hardware resembles that of a network. Consider for instance the cache coherence protocol which is realised as an exchange of messages over a complex interconnect topology. Structuring the OS in such a manner offers not only performance benefits but makes it easier to reason about the system as a whole and facilitates adapting to new hardware developments, for instance heterogeneous and non cache-coherent machines.

This is in stark contrast with conventional monolithic operating systems in which all services (e.g. drivers, file system) are implemented as part of the kernel. These are built as a shared memory program which spans the machine along with global data structures protected by coarse-grained locks. Monolithic operating systems date from the single-processor era and increasingly parallel hardware has revealed performance problems

due to contention. Overcoming this has involved considerable effort to replace with more fine-grained sharing.

Another point in the design space is the microkernel [23] which provides a minimal set of mechanisms in the kernel and traditional services running in user-space which implement a specific policy. The main OS abstractions in such a system are address spaces, threads and inter-process communication. Exokernel [12] took this idea further and separates resource protection from management. The kernel exposes just a thin interface over the hardware and the conventional abstractions are implemented in an untrusted library OS. More recently there have also been projects such as Tornado, K42 [13, 18] and Corey [6] which target multicore hardware. They demonstrate the benefits of avoiding sharing across cores through careful design of data structures.

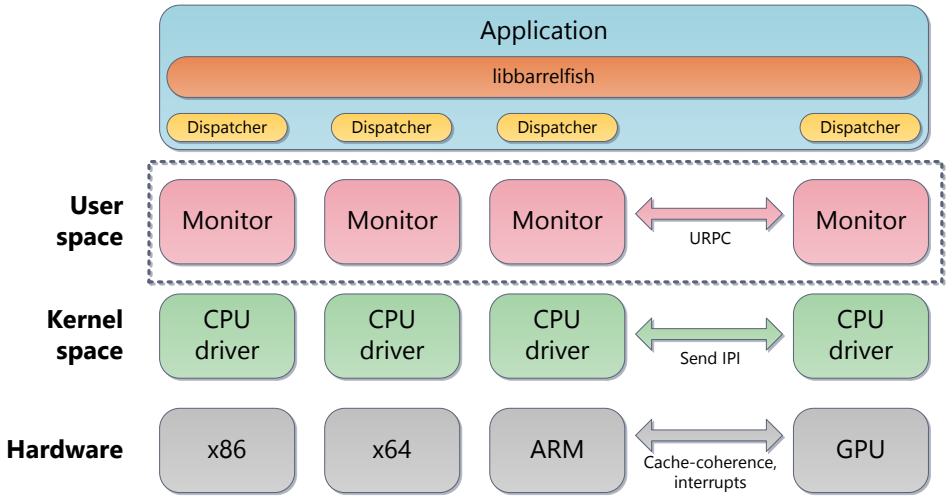


Figure 2.1: Barrelfish OS Structure

Figure 2.1 depicts the key components and the overall structure of the multi-kernel design as implemented in Barrelfish:

- *CPU driver*: This implements a limited subset of the features in a traditional kernel, namely performing privileged operations on behalf of applications and interacting with hardware (e.g. MMU, APIC, timers). Another task that it is responsible for is multiplexing the processor among dispatchers, which is done with an RBED scheduler [7] that integrates real-time and best-effort tasks. The CPU driver is written to run on a single core and is a small com-

ponent which is specialised to a particular architecture and can be easily adapted.

- CPU drivers operate in isolation on a single core. The *monitor* handles cross-core operations and replication of state, for example coordination of virtual memory mappings. This runs in user space but is a privileged process which has access to data structures such as the capabilities database. In contrast with the CPU driver most of the monitor code is not hardware specific and thus portable across machines.

- `libbarrelfish` provides a standard set of primitives (threads, memory management, notification). Applications are free to link against their own library and thereby customise the OS personality as desired. Barrelfish adopts this principle of pushing complexity and policy into user-space from Exokernel and Nemesis [12, 19].

- *Application*: Programs are statically-linked ELF binaries and code can access shared memory as usual, provided this is supported by the hardware. There are two important kernel abstractions to point out, specifically *domains* which represent an address space and *dispatchers* which are the unit of scheduling. There is a single dispatcher per core and applications can explicitly span their domain across several cores if desired.

  Barrelfish includes a realisation of scheduler activations [3, 24]. When threads block or upon allocation of the processor, the dispatcher receives an upcall and can decide how to use its time slice. This mechanism permits an implementation of threads which combines the flexibility of kernel threads with the performance of user-level threads.

- *Message passing* is one of the central themes in a multi-kernel and Barrelfish has several primitives to support this. Flounder is an interface-definition language which generates communication stubs. These provide a uniform interface for sending and receiving messages independent of the underlying transport mechanism. A server domain exports some set of functions; clients can invoke these in either blocking or non-blocking manner.

  There are several interconnect drivers which implement different transport mechanisms. LMP (local message passing) provides fast kernel-mediated communication with other dispatchers on the same core. UMP (user-level message passing) is a protocol which takes

advantage of cache coherency to transport cache lines between cores using a shared memory buffer.

2.2 SharedDB

SharedDB [14] implements an in-memory relational database which processes thousands of queries concurrently by leveraging shared computation. Incoming queries are not processed immediately but instead enqueued and processed at regular intervals. Processing an entire batch of queries at once provides the opportunity for shared execution, for instance costly table scans and joins. Through appropriate capacity planning it is also possible to bound worst case execution time: a query spends at most one cycle waiting and one being processed.

This is very different from traditional database architectures in which each query is parsed, planned and optimised in a separate execution context. This incurs sizeable costs and transactions also contend for locks which limits scalability and causes considerable cross-interaction as load on the system increases.

It should be noted that both systems have their merits and fit different use cases. The query-at-a-time model has been refined over many years and delivers high performance for a specific set of queries, provided indexes and materialised views are appropriately selected. On the contrary, batching gives consistent throughput and predictable performance even on unknown and changing workloads. This is possible because the cost of computation is shared across a batch of queries.

A key abstraction of SharedDB is that of a global query network which is a dataflow graph of relational operators. This implies there is a fixed set of queries defined ahead-of-time, although these may have arbitrarily complex predicates. The workload itself will consist of many instances of these common templates which permits sharing. An example of a query network is shown in Figure 4.1 on Page 19 which we will discuss in more detail later. Currently the query plan is built and implemented by hand, although there is ongoing work to extend SharedDB with a declarative SQL-like interface to express the schema and queries more flexibly. At runtime a cost-based optimiser builds a query plan and generates specialised code to drive the operators.

Operators receive a batch of queries, push work down the query network and conduct their processing once results are returned from these sub-

queries. Finally the processed data is propagated back up the tree. From an abstract perspective an operator can be viewed as a black box with two queues: one for incoming queries, another with results of sub-queries. There are two categories of operators: blocking and non-blocking. The former (e.g. sorting) need to materialise the full input stream before processing can begin, whereas the latter (e.g. projection or filtering) can immediately process and forward tuples as they arrive.

SharedDB achieves parallelism by running each operator in the context of a kernel thread. Most operators are single threaded, although intra-operator parallelism is employed for a few cases such as the hash join operator. Operators are bound to cores with hard affinity and this deployment remains fixed. Note that the thread-per-operator model may limit the number of cores which can be utilised depending on the size of the query network.

All data resides in main memory and (optional) durability is supported by logging all updates to disk, with regular checkpoints for recovery. There are two storage backends: Key-Value and Crescando. Tables can be accessed with index probes or full table scans. In addition a considerable portion of processing can be done directly in the storage engine, for example predicate evaluation, string matching and aggregation.

Crescando [35] is a storage engine designed to support large request rates and deliver predictable performance on unknown workloads. A large part of its scalability is due to the Clock Scan algorithm which implements a cache-aware join between queries and data. It builds an index over the query predicates and matches these against the data as it streams through. Two cursors are used: updates are applied in arrival-order followed by queries. In this way queries see a consistent snapshot of the data. Crescando also maintains a B-Tree index which is used to support scan operations and for index nested-loop joins. For especially large tables, data can also be partitioned over several disjoint segments using a hashing strategy or in round-robin fashion. In this case, a separate *controller* operator distributes requests among the scan threads and merges their results.

SharedDB is implemented in heavily-optimised C++ and makes extensive use of template metaprogramming. There is also a schema compiler which takes a table definition and generates specialised code for processing tuples. On the client side, the main logic for orchestrating an experiment is written in Java along with JNI (Java Native Interface) bindings to low-level code for serialising and sending requests. Client and

server communicate over TCP sockets using a custom protocol. An important design choice is that one thread is used per terminal; as we shall see this presented some challenges when benchmarking.

Chapter 3

Porting SharedDB to Barrelfish

The previous chapter gave an overview of the database (SharedDB) and the operating system (Barrelfish) being used in this work. In this chapter we present the modifications required and some problems encountered during porting.

The following discussion is broken down into the following sections: the first gives a broad perspective of the porting process, the second discusses specific aspects from the perspective of SharedDB and the third focuses on points related to Barrelfish.

3.1 Overview

In general the porting process consists of two stages:

- *Compile-time*: initially we had to setup the environment and deal with missing or differing functionality. These issues can be easily isolated and involve many small and straightforward changes across the project.
- *Runtime*: thereafter we needed to investigate and diagnose localised issues in specific components. By comparison, this is what made up the bulk of the effort.

Our plan was to gradually build up the port, starting with the utilities library and some unit tests. Subsequent to that would follow a small-scale Crescando demo and finally the full TPC-W workload running on SharedDB.

3.2 SharedDB

3.2.1 Project structure

SharedDB is a large codebase and it took some time to understand the key abstractions and overall structure. The papers [35, 14] are intended for the research community and accordingly focus on the algorithms and processing model. The most important interfaces (such as Table, Operator, Result) are documented and as a beginner it would help to have a guide with pointers to these.

To illustrate, here are some notes on how the different components relate to one another. SharedDB's codebase is structured as several projects which progressively build on the functionality of one another:

- *Utilities*: Wrappers over system-specific functions (memory management and threading) and primitives such as logging, queuing and reference counting.
- *Crescando*: Implementation of the storage engine. More specifically, this contains the logic corresponding to the scan operation, indexing, statistics, durability and partitioning. Clients make use of this via a generic table interface which provides functions to store tuples and execute queries.
- *Schema compiler*: Takes an SQL-like table definition as input and generates schema-specific code to process data within Crescando.
- *SharedDB*: Implements the relational operators (e.g. joins, grouping and sorting) and provides a unified interface over the storage engines (Crescando, key-value).
- *TPC-W driver*: Server-side code which implements a specific workload. This consists of several schema files and code to build the query network, populate tables and query logic to drive the operators.

3.2.2 Build system

The existing build system is based on the GNU Autotools suite. Although this tool includes support for cross-compiling, more complex scenarios involving code generators require additional attention. We encountered the problem that the Crescando schema compiler is built for the host platform (in our case Barrelfish) and therefore fails when executed on

the native platform during the build process. Beyond this, we needed all code to be statically linked with the main executable.

Barrelfish's build system (Hake) is designed around similar needs, however it is not aware of the external toolchain and would require further modification. We therefore opted to switch to CMake with which we had prior experience. CMake has support for adding custom commands to the build graph and we wrote a macro to encapsulate the dependencies and actions needed to run the schema compiler. The primary advantage this offers is that the entire codebase can be built in a single invocation and when dependencies are modified all appropriate targets are rebuilt automatically. Furthermore, built artifacts are laid out in the appropriate layout to facilitate deployment to the cluster and we could also support several build configurations from a single source tree.

3.2.3 Static linking

Crescando generates code at compile-time which is specific to a schema definition and compiles this to a shared library which is loaded at runtime. A function named `csd_storage_table_create` is then used to create an instance of a specific table. Dynamic linking is not currently supported on Barrelfish so we needed to make some modifications to support static linking. The first change was to resolve conflicting symbols by including the table name in this function (e.g. `csd_storage_author_table_create`). A second change added an interface to transparently use the schema-specific libraries in a portable manner. Based on a configuration macro this will either call `dlsym` or return a fixed function pointer depending on whether dynamic or static linking is used.

3.2.4 Modifications

SharedDB is developed on Linux and depends primarily on the standard C and C++ libraries and a small set of POSIX functions, such as pthreads and BSD-style sockets. In addition, it makes use of Boost which is an extensive C++ framework to complement the standard library. Only a select few header files are needed and no code changes were necessary. There is also an optional dependency on `libnuma` although this was omitted from the port.

Barrelfish uses newlib as C standard library and complements this with `libposixcompat` which offers a limited set of POSIX interfaces to ease porting existing applications. The vast majority of the func-

tions SharedDB needs were already provided by these libraries. A few Linux-specific functions are used in SharedDB but these could be easily replaced with an alternate implementation. As an example, we broke the dependency on the system type `CPU_SET` - which is used extensively throughout - and substituted those usages with a portable CPU affinity mask. Aside from this there were two other classes of changes. The first added feature tests and macros to disable features which are not available, for instance the socket option `SO_REUSEADDR` or structs with differing fields. A second group of changes was needed to fix missing `#include` directives. The tool *include-what-you-use* [16] was used to make these changes automatically.

In order to have some assurance of correctness we also wrote and ported some unit tests. In retrospect these had limited utility because they only cover a small portion of the basic utilities and some Crescando data structures. Although the unit tests helped to isolate one or two issues, the more complex problems we encountered were better served by full end-to-end integration tests. A sample issue we encountered is stack overflow. SharedDB often makes use of `alloca` and places structures on the stack (e.g. trie for prefix queries). This happens because of the small default stack size on Barrelfish (64 KB) compared against Linux (8 MB). This is easily remedied by allocating a larger stack. Nonetheless, we added a warning to the page fault error which has been especially helpful in other situations.

Overall not many invasive code changes were needed to support Barrelfish and those which were are largely confined to the system portability layer and network I/O. We discuss the latter in more detail shortly.

3.3 Barrelfish: Porting

3.3.1 C++ support

Barrelfish is not self-hosting. Instead you develop on a remote machine, cross-compile to static binaries and then run on a simulator or deploy onto a rack machine. The build process makes use of the host's C compiler. This works when compiling freestanding C programs because Barrelfish provides its own libraries and startup objects.

For C++ we made use of a cross-compiler which had been previously ported to target Barrelfish and updated it to the latest version as needed by SharedDB. The toolchain port itself entails modifications to GNU binu-

tils, which provides the assembler and linker, and GCC (GNU Compiler Collection) for the compilers and C++ standard library. All the important language constructs work including virtual functions, templates, thread locals and inline assembler. Exception handling does not work but this was not a hindrance because SharedDB does not make use of this language feature.

Whilst C and C++ are mostly compatible at the source level, minor changes were required to Barrelfish's header files. C++ is more strict about function casts and does not support some C99 features, for instance designated initialisers. In addition, function definitions needed to be wrapped in `extern "C"` blocks to demarcate the linking mode.

In Barrelfish each subsystem, such as the file system layer (VFS), needs to be initialised before usage. SharedDB's logging system reads a configuration file during its initialisation, however, this is triggered by constructors of global objects which run before the `main()` function. Individual functions can be decorated as constructors with an integer to indicate their priority. Using this we could arrange for VFS initialisation to occur at the appropriate point.

3.3.2 Synchronisation primitives

Barrelfish's threading library provides mutexes, condition variables and semaphores but was lacking barriers and thread-safe initialisation ('once'). These routines were easily implemented. What was a little more involved is support for blocking waits with a timeout, which we now describe briefly. This functionality is used within SharedDB to allow for clean shutdown. More importantly though, lwIP (the network stack) uses this as the basis to portably implement timers.

The existing synchronisation primitives have functions to block a thread and wait indefinitely until it becomes ready, but in this instance what was needed is the ability to re-awaken the thread if a timeout elapses. Deferred events are not appropriate for this purpose because the thread needs to run an event loop but it is blocked. We added an analogue mechanism which is triggered automatically by the dispatcher upcall. If a wait timeout expires, the corresponding thread is removed from the queue where it is waiting and added back to the runnable queue. Conversely, if the thread is awoken in the interim its deferred event is cancelled. All of this involves manipulating doubly-linked lists which can be done in constant time.

The implementation of threads in Barrelfish is very elegant, minimal and straightforward to understand. Having this implemented in a user library makes it easy to prototype new features which underscores a benefit of the library OS design.

3.4 Barrelfish: Diagnosing

3.4.1 Library OS design

In keeping with the microkernel design Barrelfish implements a large portion of its services as unprivileged code in user-space. Examples of this include device drivers and the network stack. When porting SharedDB we had no concrete plans to tailor the OS flavour, however simply having this possibility has been useful in a number of situations.

One major benefit is that this leads to a structure with many self-contained modules which can be readily understood and modified. This makes it considerably easier to troubleshoot unexpected behaviour due to the ability to add custom instrumentation. For example the dispatcher upcall can be used to implement a rudimentary profiler. To give another example, we also encountered an implementation-specific issue which prevented allocating more than a few gigabytes of memory. Barrelfish has an extensible virtual memory system which runs in user-level modules and so our application could be modified to request memory pages from the system and map them into its own address space.

3.4.2 Network stack

Barrelfish's network stack consists of an Ethernet driver, several services (queue, port and filter managers) which run as separate domains and lwIP which is linked against applications. The latter implements the TCP/IP protocols and interested readers are directed to [11] which describes its design and implementation in further detail.

At the edge of SharedDB's query plan are server operators which interact with external clients. These are written using conventional BSD-style sockets with blocking calls and a `select()` loop. Although this interface is offered by lwIP, we encountered stability issues at load and when used from several threads. To resolve this we instead modified SharedDB to use lwIP's event-based 'raw' API. The major change this involved is that all network operations need to occur from a single thread. The server

operators therefore call stubs that forward processing to a centralised thread which performs the operations on their behalf. This potentially involves cross-core communication which could affect performance and is something we may need to re-visit later.

3.4.3 Summary

This section presented some of the work involved in porting SharedDB to Barrelfish. Whilst there were some challenges and modifications needed these were primarily implementation issues and using existing facilities in scenarios where they had not previously been tested. Nevertheless, the design of both systems complement one another well and they will serve as a good foundation for further co-design. As of the submission of this thesis, the port of SharedDB to Barrelfish is functional but there is a minor unresolved issue related to cross-core messaging. We expect this problem to be resolved soon, and will compare its performance characteristics against a run on Linux. We do not foresee any bottlenecks although there may be differences in performance due to the network stack and scheduler. In the next chapter we establish an initial baseline of SharedDB running on Linux.

Chapter 4

Linux Experiments

4.1 Introduction

At this point we understood the inner workings of SharedDB, however we lacked an intuition for its performance characteristics. SharedDB departs radically from the design found in conventional databases and in fact may fare better under load because it is designed to leverage sharing among queries. This section describes a series of experiments we conducted on Linux to quantitatively analyze its behaviour and better understand its performance under a number of different configurations.

In particular we were interested in investigating questions such as the following:

- *Scalability*: What happens as the number of cores is varied? When there are more cores than operators does it matter which subset is used? On the contrary, how does the system react to the processor time being sliced among several operators?
- *NUMA awareness*: SharedDB makes provision for NUMA architectures in its placement of operators and when allocating buffers. To what extent does this boost performance?
- *Deployment*: SharedDB explicitly pins threads and has a handcrafted mapping of operators onto cores. A natural question to ask is why this is done and how it affects performance.
- Which portions of the query network are most active? Further, is it possible to characterise the resource usage of each operator?

In the coming sections we shall first provide an overview of the environment and then zoom in on each question, present our expectations and discuss the results we found. As we shall see, on our specific workload SharedDB already achieves its peak performance after exhausting only a small fraction of the resources available on a large multi-core server.

4.2 Methodology

4.2.1 Workload

TPC-W [25] is a standardised benchmark set in the context of an online book store which implements a typical three-tier application architecture. Web browsers, which assume the role of customers, interact with the store by browsing, ordering and shopping for books. These are sent to a web server which in turn generates requests to the data layer. These requests are made up of primarily point queries and short-running transactions.

The specification defines fourteen types of interactions which can be broadly classified as either *browsing* (searching and requesting details of products) or *ordering* (adding items to the cart and purchasing). Beyond this, there are three *workload mixes* which govern the proportion of each interaction type effectively sent to the system: browsing, shopping and ordering. The first has a breakdown of 95% browsing to 5% ordering which makes it mostly query-intensive. By contrast the last has an equal split between the two categories and consists of more updates. The middle has an 80-to-20 ratio and is a compromise with a mix of queries and updates.

Figure 4.1 shows the query plan corresponding to the TPC-W workload in SharedDB. At the base of the network are 10 storage operators whose tables store the details of the inventory, customers and their orders. There are 12 server operators at the edge of the network which communicate with client machines and the query logic is realised by a further 16 relational operators. You may notice these perform joins, sorting and grouping but there is a distinct lack of primitive filtering operations such as `LIKE` and `LIMIT`. These operations are pushed down and evaluated directly inside the storage engine together with predicate matching.

There are two different storage engines in use: Crescando is used for the Address, Author, Country and Item tables; the remainder are backed by key-value stores with indexes on the key attributes. Two-way partitioning is also employed on the Address and Item tables.

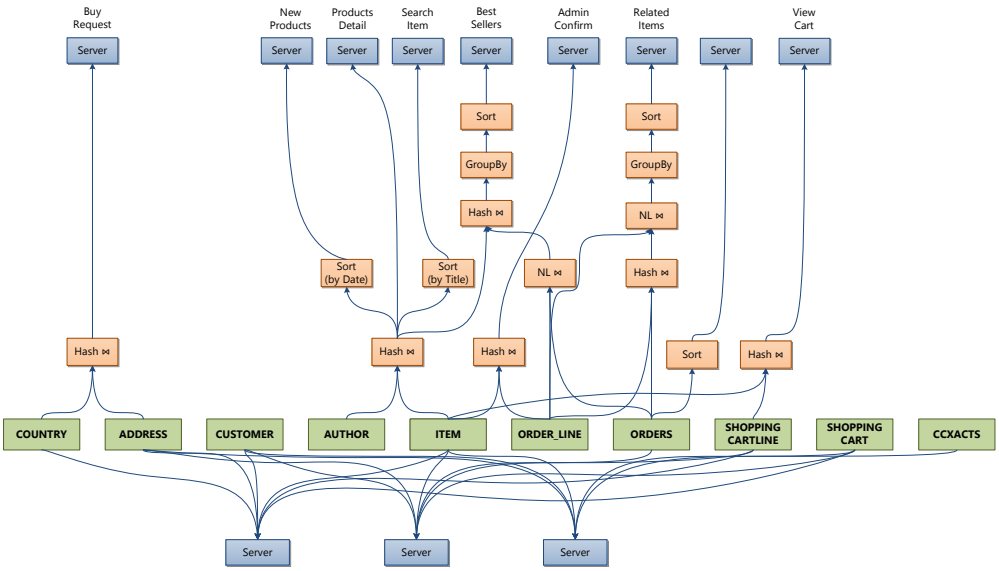


Figure 4.1: TPC-W: Global Query Plan

4.2.2 Metrics

There are two primary metrics we consider during the experiments:

- Median response time (milliseconds)
- Throughput (WIPS: web interactions per second)

In our context, response time is measured end-to-end from the client perspective – i.e. from the time a request is sent until the full response is received including network latency. The metrics we present are aggregated over all clients, but there remains the option to zoom in to a specific transaction type if needed.

Each query has an associated time limit. These are generally between 3 and 5 seconds with two exceptions: 10 seconds (Search Results) and 20 seconds (Admin Confirm). Responses are considered valid as long as their 90-th percentile latency does not exceed the given limit for that transaction type.

4.2.3 Factors

The following is a non-exhaustive list of the factors influencing our experiment:

- Load (number of emulated browsers).
- Placement of operators onto cores.
- Total number of cores allocated.
- Operating system.
- Network topology.
- Hardware (in particular caches and interconnect structure).
- Workload mix.

We will thoroughly investigate the first three factors and keep the remainder fixed. Given that there are interactions – for example between deployment strategy and hardware – each experiment will target a single factor. Unless mentioned otherwise we used the default placement strategy and all the cores on the machine.

4.2.4 Environment

In the subsequent experiments we used a single machine to host the database server and twelve client machines to generate load. Their specifications are as follows:

- *Server ("appenzeller")*: quad-processor, 12-cores (AMD Opteron 6174 "Magny Cours" running at 2.2GHz), 128 GB RAM over 8 NUMA domains, 64 KB L1-d, 512 KB L2 and 12 MB shared L3 cache.
- *Client ("dryad")*: 8 cores (AMD Opteron 2376 "Shanghai" running at 2.3GHz) and 16 GB RAM.

The server and client machines are not attached to the same switch but are rather separated by three network hops with average round-trip time of 0.22 ms.

The code itself is compiled using GCC 4.7.2 at optimisation level 3, for the native architecture and with assertions disabled. The runtime environment is consists of 64-bit Debian "Sid" running Linux kernel 3.6.10 and glibc 2.13.

For the purposes of these experiments all data was held in main memory and durability was disabled so there was no disk I/O involved. The cardinality of the tables is controlled by two factors: number of EBs (emulated browsers) and number of items (1K, 10K, 100K, 1M or 10M). The actual values used were 100 EBs and 10K items respectively, which

results in a total dataset size of less than a gigabyte. These numbers may seem small relative to the size of main memory, but recall that SharedDB's primary use case is sharing which is especially effective when processing a vast quantity of queries in parallel. There is nothing inherent in its design that would prevent use of larger datasets and this is the subject of ongoing experiments.

4.2.5 Modifications

There are a few specific aspects of our experimental setup worth nothing as they deviate from the specification:

- Given that we are interested in benchmarking SharedDB, we will exclude the web server layer and clients will instead issue their requests directly to the database. For the purpose of our experiments the system-under-test (SUT) is defined as the database in isolation. For comparison, TPC-W defines the SUT rather broadly to comprise all the web servers, databases and network infrastructure used in the implementation.

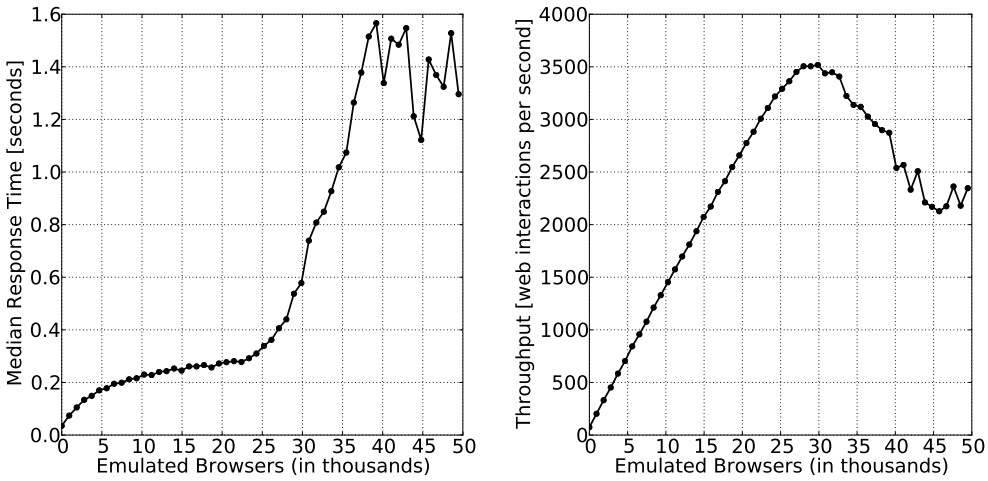
- To reduce the scope of our experiments we will only consider a single workload mix: browsing. The ordering mix has a larger proportion of updates and primarily stresses durability and the disk I/O subsystem. This is a reasonable choice because it involves complex queries with several table scans, joins and sorting/grouping which exert a considerable portion of the query network.

- Emulated browsers wait for a short interval between the arrival of a response and sending their next request. This is referred to as the *think-time* and is defined to be Poisson distributed with an average of 7 seconds. One specific quirk we needed was to scale this by a factor of roughly 0.6, which reduces the time between interactions and consequently has the effect of simulating more emulated browsers. This was necessary because each terminal runs in the context of a Java thread and we observed undesired garbage collection effects with a few thousand threads.

4.3 Results

4.3.1 Baseline

Our first experiment attempts to reproduce the results from the VLDB'12 paper [14]. Based on those figures we expected to reach peak performance of about 1500 WIPS with 12000 emulated browsers (on the same hardware) and deployed accordingly.



(a) Response Time (b) Throughput

Figure 4.2: Baseline Performance, Varying Load

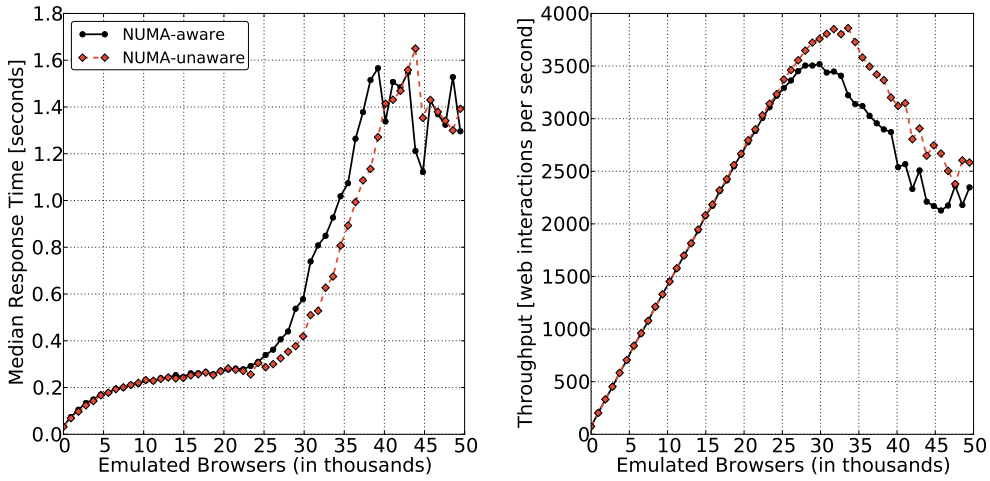
The results we obtained are shown in Figure 4.2. We observe that throughput rises steadily until roughly 27000 browsers and the system is able to cope with the increasing load. Response times also increase very gradually and the median response time remains below half a second until this point. Beyond this threshold the system becomes unstable and exhibits a decline in throughput and fluctuations in response time. This is largely attributable to the browsing workload mix which has several heavy search interactions. In particular the Best Sellers query involves several joins, sorting and grouping and the system spends time processing these but responses eventually exceed their time limit.

To summarise, SharedDB scales in proportion to the load and is able to do so because it shares processing over a large number of concurrent queries.

4.3.2 NUMA awareness

SharedDB includes some logic for NUMA hardware (described in more detail in Appendix A.1). We will now conduct an experiment to compare whether this affects performance.

Hypothesis: At this stage it is unclear what quantity of data is shared between operators and whether these accesses could saturate the interconnect. In the extreme case of data shuffling NUMA awareness can yield a dramatic improvement (3x) [22]. In SharedDB's setting, however, the overall effect on our two metrics is likely to be on the order of several percent because these effects are minimal compared to the cost of network transmission.



(a) Response Time (b) Throughput

Figure 4.3: NUMA-awareness Comparison, Varying Load

The results we observed are shown in Figure 4.3 which compares performance with NUMA-awareness both enabled and disabled using the hand-tuned deployment strategy. Counter-intuitively *disabling* NUMA awareness actually gave better performance beyond 25000 browsers: peak throughput increases by just under ten percent. The reason for this behaviour is not clear, although it may be attributable to the deployment strategy which was devised by analysing the communication patterns between operators. Linux has a policy of allocating memory on the NUMA node where it was first touched. Another possible explanation is that this results in better memory placement than the hard coded policy

of allocating on a specific NUMA node as SharedDB implements. In any case, we did not investigate further but simply omitted this aspect from the Barrelfish port and can revisit this decision if we observe a performance anomaly.

4.3.3 Deployment strategy

In this series of experiments we extensively investigate how the placement of operators onto cores affects performance. We will compare three different policies: hand-tuned, "largest-first" heuristic and not pinning threads. As a second dimension we will vary the number of cores and explore how the database behaves as resources become more scarce.

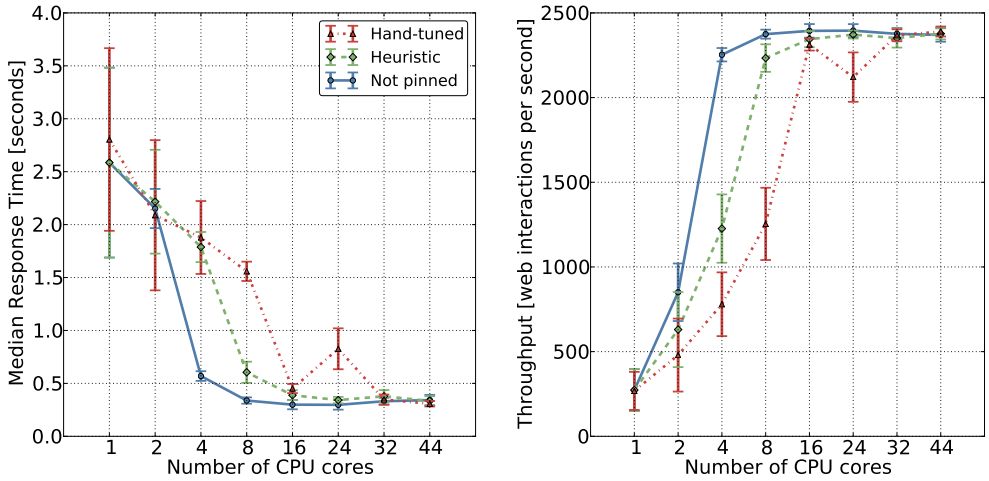
SharedDB's hand-tuned deployment strategy was devised by analysing access patterns and data transferred between operators and we expect it to perform well. It incorporates the heuristic of scattering Crescando scan threads and gathering SharedDB operators. The reasoning behind this is to have good locality between operator queues by placing scan threads as 'far' apart and thus minimise conflicts in the shared last-level cache.

Operator placement is specified in a configuration file which defines the core number for each operator. The effective core is derived by taking these numbers modulo the actual number of cores. As we shall see, in specific cases performance is degraded with the hand-tuned strategy due to conflicts between sensitive operators. To address this we devised a simple strategy which places large table scans and critical join operators first. When devising this we considered the number of tuples passing through and the CPU usage of each operator.

Finally we also compare against not setting hard affinity on operator threads and instead relying on the operating system's policy. We refer to this deployment as `no-pin`.

For this experiment we used a fixed load of 18000 browsers, varied the number of cores and tried different deployment strategies. The results are presented in Figure 4.4. All three strategies perform identically when using the maximum number of cores [1]. However, as the number of cores is reduced we observe three distinct trends corresponding to each deployment strategy. The hand-tuned strategy performs worst overall,

[1] Although the machine has 48 cores, the query network only has a total of 44 operators so a few cores are unutilised.



(a) Response Time (b) Throughput

Figure 4.4: Effect of Deployment on Performance

our heuristic improves on this and surprisingly not pinning threads yields a striking improvement. In particular take note of the steepness of the respective curves.

As expected, performance decreases when there are fewer resources allocated but the point at which this occurs is of note. When leaving decisions to the kernel, peak performance can be reached using just 8 cores and beyond this the situation does not change. Viewed differently, using only one tenth of the cores gives just a 5% decrease in throughput. The same effect can be observed with the other deployment strategies although less pronounced. This may be partly attributable to the size of workload we use. A further explanation for this behaviour is that there are a few active operators and the remainder are comparatively dormant. This is reflected in the figures shown in Table 4.1 which shows mean CPU usage of the top five operators in descending order.

The issue with the default deployment strategy is that it treats core numbers as abstract values whose order doesn't matter. Consider the data point with 24 cores: the Orders table is co-located with one segment of the Address table, which is an instance of an undesirable conflict because both operators are hot. Our heuristic assigns operators largest-first which helps mitigate bad cases where two hot operators are forced to share a core. This is clearly reflected by the more stable behaviour and overall higher throughput. A typical justification for explicitly pinning

Operator	Mean CPU usage (%)
Address (Crescando, Segment 2)	84.3
Address (Crescando, Segment 1)	77.6
Item (Crescando, Segment 1)	61.6
Item (Crescando, Segment 2)	57.9
Author $\bowtie$ Item	26.2
...	
Total	11.5

Table 4.1: Top-5 Operators by CPU Usage

threads is to avoid potentially costly thread migrations which thrash the cache state. Our experiments demonstrate that in SharedDB's case this is unwise and actually degrades performance. Leaving decisions to the kernel gave the highest throughput and lowest response time in all cases. It is challenging to gain insight into the kernel's scheduling decisions, but we believe this is due to having full flexibility to re-arrange threads and quickly react as load fluctuates.

Aside from the spatial aspects we have explored, this experiment shows what occurs when the cores are over-subscribed. It is not surprising that performance suffers because several threads now compete for the processors' time. When using just two cores a third of the queries being processed exceed their time limit and consequently throughput halves and response time quadruples. This underscores the importance of appropriate capacity planning.

This section explored the effect of operator placement on performance. We found that deployment strategy does indeed matter, although it was best not to set hard affinity but rather leave these decisions to the Linux kernel. Further, our experiments show that on our specific workload resource utilisation is low and it is possible to achieve peak performance with less than a quarter of the resources!

4.3.4 Operator characterisation

In the previous section we established that resource utilisation is low and we want to delve further and understand why this is the case.

Merkel et al. [26, 27] proposed the notion of *Activity Vectors* in prior research in the context of energy- and temperature-aware scheduling. These vectors are associated to a task and express the extent to which

functional units within a processor (e.g. TLB, ALU) are utilised. They are derived using performance monitoring hardware in the processor, updated at regular intervals and are used to influence scheduling decisions.

Our aim is to apply the same technique to operators in the TPC-W query network to characterise their resource usage and potentially identify bottlenecks. We expect that operators can be broadly classified as CPU-, IO- or memory-bound and if so, this should be reflected in their activity vectors. Such a classification could be used to derive a suitable deployment strategy when there are more operators then cores. To measure resource usage we will use performance counters; Appendix A.2 provides additional background material.

In this series of experiments we use a fixed load of 18000 browsers and a deployment with no overlap. No two operators share the same core such that their activity can be observed in isolation. We record performance counters and our primary areas of focus are: CPU utilisation, memory bandwidth and cache utilisation (private and shared). The output is a matrix with a vector for each operator and where each vector consists of twelve dimensions.

We faced some issues when trying to interpret this data because there are many dimensions with few discernible patterns. Figure 4.5 shows a heat map of the correlation matrix. The majority of the dimensions are highly correlated and exhibit little differentiation. In an attempt to make sense of these figures we applied dimensionality reduction techniques such as PCA (principal component analysis) and k-means clustering. This analysis revealed that the dataset has low intrinsic dimensionality; 99.93% of the variance can be explained with two components. The issue, however, is that the actual direction of these components point somewhere in the space. As such the projection of our data onto these directions does not relate to the original metrics so it cannot be interpreted directly. Visual inspection showed two clusters – 'hot' and 'dormant' operators respectively. The hot operators we previously mentioned exhibited high CPU usage, as expected, but in other dimensions such as cache miss rate they did not exhibit much variation.

In summary, these experiments did not provide the insight we had hoped for. It may be that a more thorough investigation of a larger set of performance events is needed, in conjunction with more sophisticated analysis techniques.

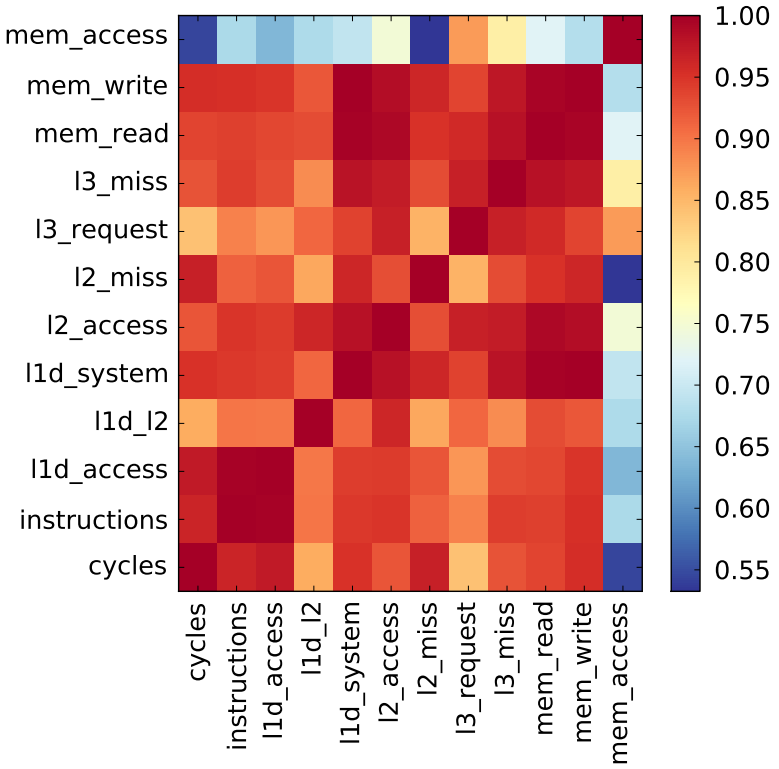
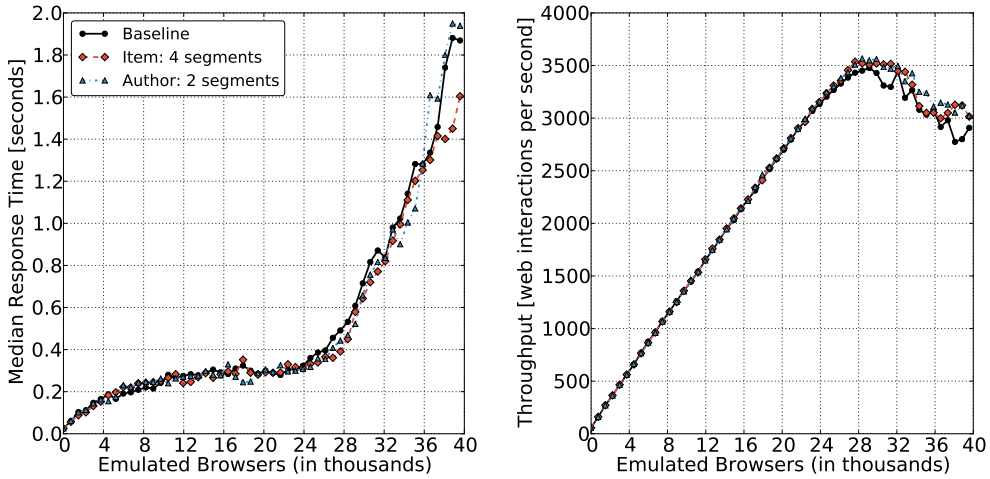


Figure 4.5: Correlation heat map for performance counters

4.3.5 Partitioned tables

As we observed in a prior experiment, resource utilisation is rather low yet we would like to leverage the full potential of the machine. One possibility to overcome this is to focus on bottleneck operators or replicate several copies of the entire query network. The former can be easily realised by partitioning hot tables which we try in this experiment.

The results are depicted in Figure 4.6. Although this does result in a throughput boost of a few percent, it is not as promising as initially hoped. Tuples are distributed in a round-robin fashion over the segments of a table and a central controller handles operations across partitions. Subsequent investigation revealed that this operator is reaching the so-called 'batch limit'. This is a knob which adjusts the granularity of requests (queries or updates) which are passed around as a unit. The choice of this parameter is guided by a trade-off: larger values can improve sharing, but this should also be small enough that the tuples



(a) Response Time

(b) Throughput

Figure 4.6: Partitioned Crescando Tables, Varying Load

fit into the cache. Resolving this is not within the scope of this work, although it would be interesting to deploy several replicated engines with a middleware layer for coordination, as in Multimed [30].

4.4 Summary

In this section we presented a series of experiments intended to explore specific aspects of SharedDB's performance. To a large extent it behaves consistently with the behaviour we expected. Starting with the baseline, we observed that SharedDB is able to sustain a large number of clients at high throughput because of shared computation. A surprising finding was that on our specific workload resource utilisation is low and it is actually possible to achieve peak performance using only a small number of cores. Digging deeper, we tried to quantify each operators resource usage using performance counters and clustering but struggled to gain meaningful insight. Finally we tried table partitioning in the hope of boosting throughput but this brought limited gain. We believe this is due to limitations of the current implementation.

Chapter 5

Co-Design

The following section discusses two interesting questions which arise when consolidating operators onto fewer cores and concludes with a discussion of how applicable it might be to run SharedDB on a non-cache coherent architecture.

5.1 Operator consolidation

When running the TPC-W benchmark SharedDB is able to deliver high throughput and sustain tens of thousands of clients, but resource utilisation is low on our particular workload. Our experiments demonstrated that it is possible to consolidate to a smaller number of cores without severely impacting performance. In this context there are two interesting sub-problems related to operator scheduling.

- *Spatial deployment*: In our experiments we observed that different operator placement strategies have an impact on performance (see Section 4.3.3). This theme investigates how to group operators onto cores. To illustrate this, there are some operators which are cache-sensitive that could interfere, and others which could benefit from co-location because they share a working set residing in cache.

  This problem is outside the scope of this thesis and we will not describe it in more detail, but instead we summarise the approach. Initially the database is run across all cores to gather performance counter measurements which characterise the operators. These are subsequently fed into an optimisation algorithm which applies bin packing. The result is a deployment plan which uses fewer re-

sources and still meets the application's SLA. In a limited sense this is similar to the knapsack problem of job scheduling in datacenter deployment.

- *Temporal scheduling*: When using 8 instead of 44 cores, our results showed that total throughput is reduced by less than 5% however response time increases by roughly 30% and exhibits higher variability. We believe this is attributable to time sharing and the operating system's scheduler not being aware of the application's requirements. Databases are aware of data dependencies and can estimate the resources required to process a set of queries. The question we pose is whether the database perform better scheduling decisions based on the knowledge it has.

Due to the limited scope of this thesis we were unable to implement and evaluate these ideas, however we will define the problem and present initial thoughts.

5.1.1 Temporal scheduling

Consider the problem of time slicing among several tasks. This can be broken down into three main components:

- Which task next?
- For how long?
- On which processor?

A simple scheduler may use a round-robin strategy and time slices on the order of tens or hundreds of milliseconds. Modern operating systems are more sophisticated and have variable time slices, dynamic priorities and more complex scheduling disciplines which strive to make decisions in constant time. Additional details on Barrelfish's scheduling infrastructure can be found in [29].

Operating system schedulers need to support a mix of workloads including throughput-oriented batch jobs and interactive tasks which forces them to make trade-offs. Consider an operator which is pre-empted at an inopportune moment; this could pollute its working set and increase its runtime. Some of the performance degradation we observed may stem from poor scheduling decisions due to lack of coordination between OS and database. We believe that by using application knowledge, such as data dependencies and task estimates, it may be possible to improve on

this situation where the core is shared between several operators. An initial goal would be to confirm that smart scheduling decisions can, for example, reduce end-to-end response time for a query.

Problem definition

Inputs

- *Spatial deployment*: Static mapping of operators onto cores.
- *Runtime statistics*: Databases have cost models which can provide an estimate of the execution cost for a batch of tuples. They also maintain information such as operator selectivity, number of tuples materialised, etc.
- *Query network*: Standard boxes and arrows model from data-flow systems. More concretely this is a directed acyclic graph where each node represents a database operator and edges indicate a data dependency. Fan-out: each node has exactly one parent and either zero, one or two children. Examples of each are leaf table scans, sorting/grouping and joins respectively. There are no constraints on the depth of the query network but typically this will have no more than 5 levels.

Output Using the inputs above, the algorithm should return a *scheduling plan*. That is a serialisation of the tasks on a particular core which is optimized for some metric, perhaps latency or throughput. By tasks we refer to operations such as a single table scan or the build and probe phases of a join operator.

Simplifications As an initial starting point there are several simplifying assumptions which can be made to reduce the problem scope:

- No external applications are allocated to the same set of cores, so the scheduler has full visibility and control over what is happening.
- Operator placement remains static and each scheduler is only responsible for a single core. In this way one does not need to consider thread migrations.
- Only consider pairwise dependencies between neighboring operators, which considerably prunes down the multitude of possibilities.
- There needs to be some constraint on the time period during which operators may potentially be active, such that an ideal scheduler

could actually fulfill the work. The spatial deployment will generate a reasonable placement such that no core is at peak overload.

Micro-benchmark

When switching between tasks there are two different costs incurred. There is the *direct* cost of running the scheduler and dispatching to another task, which involves saving and restoring the processor state. There are also *indirect* penalties due to effects such as pollution of caches.

In order to gauge whether our approach is viable we would need to gauge the magnitude of these costs and assess the ratio of useful work to wasted time. One approach would be to conduct an experiment which varies the time slice between two extremes: regular tick and no preemption (co-scheduling). The aim would be to demonstrate a scenario where either extreme affects performance and thus support the claim that a compromise is needed.

Prior work in this area [21, 33] has shown that working set size and access stride play an important role, so it would make sense to select cache-sensitive operators (e.g. Crescando scan) for full effect. Dispatch latency figures alone would not be too informative and it would make sense to instead measure the overall time for task completion on a representative workload. It is expected that co-scheduling will perform best since there is no interference between tasks, but this will also affect system responsiveness because other tasks cannot make progress in the interim.

Related work

In [8], Carney et al. present work on operator scheduling from the realm of data stream processing in Aurora. This makes a good comparison because it shares similar requirements of supporting large volumes of data with low latency. These techniques are also largely applicable to databases, for instance the traversal models which optimise for throughput, latency or memory. The paper also presents benefits of batching of operators (superboxes) and tuples ('trains'). The latter closely resembles the query processing model of SharedDB. The use of QoS specifications to influence scheduling decisions is also intriguing because it permits the system to shed load when demand exceeds capacity.

Another class of programs which implement their own scheduling are parallel runtimes. These are intended for different scale where there are a large number of fine-grained and possibly short-lived tasks. An example

thereof is Erlang which has a scheduler per core with work stealing. The aim here is for low latency with soft-realtime guarantees. It achieves this with accounting by assigning a reduction budget and by having separate per-process heaps. Further information can be found in [36, 2].

5.2 Non-cache coherent architectures

Barrelfish's design with explicit messaging lends itself to non-cache coherent architectures such as the Intel SCC (Single-chip Cloud Computer). In this section we provide some perspective on what might be involved to have SharedDB running on such a system.

Processors typically found in computing devices today are *cache-coherent*. This refers to the fact that the contents of the caches are kept consistent between all processors by means of a hardware protocol. Scaling such a memory architecture is challenging as the number of cores in a chip increases, so hardware designers are investigating alternative designs where the individual cores are connected via a fast network fabric. This has implications on software which has long been able to assume a shared address space and now needs to employ message passing.

The design of Barrelfish and SharedDB are similar in that they both adopt a philosophy of minimising shared state. Where they differ, however, is that the former assumes no sharing at its bottom layer, whereas the latter is written as a conventional shared memory program. To illustrate this, consider how results are passed around – as reference-counted pointers over synchronised queues. From a structural perspective though, sharing is not necessary because operators work independently of one another and exchange data in a well-defined manner. As such, it should in principle be possible to run SharedDB on a system without cache coherency.

As a starting point it could make sense to adopt a multi-process design and place each operator in a separate domain. Having several address spaces would immediately flag any sharing issues. Operator queues provide a typical get/put-style API which could be easily implemented over messages. What could prove more challenging is the initialisation code, which populates tables with data, builds the query network and attaches operators to their neighbours. This is easiest to express in a single piece of code which has access to all the required entities, however it could be broken down into localised operations. One potential downside of carving SharedDB into standalone units is that there is no longer a

single point of control, which could make it more challenging to apply techniques such as co-scheduling.

Communication is likely to place a central role in such work. On a shared memory architecture, operators in SharedDB exchange results by passing around pointers. A non-cache coherent version may need to pass the actual tuples and for large datasets this could place a strain on the shared interconnect. Some queries may have strict SLAs and it could be interesting to explore whether ideas such as the network calculus could be applied to provide QoS guarantees. There may also be some interesting communication patterns which arise, for example sending the results of a shared computation to several operators could be viewed as multicast. Updates on replicated tables involve consensus algorithms. The existing literature discusses how to efficiently implement this over a network and there is the question of whether this is also suitable within a single machine.

It is imaginable there may be hybrid architectures with 'islands' of cache coherence. Parallelised operators could benefit from this by using shared memory to coordinate their work. Further, on such machines one could adopt an approach similar to Multimed [30] with several replicas each running SharedDB in shared memory on a group of cores and middleware to coordinate execution.

Chapter 6

Conclusion & Outlook

6.1 Discussion

The goal of this thesis was to investigate the topic of co-design in the context of two research systems: an operating system (Barrelfish) and a database (SharedDB). The main accomplishment is to bring the two systems together and explore potential synergies. SharedDB is a large application which exploits modern hardware and has demanding requirements. In particular, on the TPC-W workload we used, it is capable of supporting tens of thousands of clients, thousands of concurrent transactions each second and response times of several hundred milliseconds. This makes it a good use case to improve the collaboration between database and OS and as a basis for further research. In this work we introduced the problem of consolidating resources, whilst retaining the performance characteristics of the system and presented some approaches using spatial and temporal scheduling.

The porting process presented some challenges and required changes to both systems, such as modifying the memory management and dealing with limitations of the network stack. All together, though, these were mostly issues of the current implementation and do not point to problems of the model itself. In addition to porting, we also conducted a series of experiments to quantify aspects of SharedDB's performance on Linux. The results we presented indicate that NUMA awareness and explicitly pinning threads actually diminishes performance. Further, we found resource utilisation to be low – peak performance could be achieved with just 8 of 44 cores. This could be due to our choice of TPC-W workload and is the subject of further investigation. We also presented some

work to characterise operator resource usage using performance counters although this did not provide the insight we hoped.

To recap, we believe this work mostly addressed our initial motivation and provides good potential to further the collaboration between databases and operating systems and improve resource management. The systems used in this thesis make a good fit because they both adopt a shared-nothing design. Beyond the scalability benefits which have been extensively discussed in the literature, the resulting implementation is simple, elegant and thus easy to reason about which is especially helpful when troubleshooting.

6.2 Future work

Now that SharedDB runs on Barrelfish there are numerous avenues worthy of additional investigation. Having full control over the entire stack and extensive experience in the group is also beneficial. Together, this enables testing of new ideas for co-design and optimisations which bisect several layers of the system.

- Once the problem with messaging between cores is resolved, the next step is to conduct a thorough investigation of SharedDB's performance when running on Barrelfish. We do not expect any major differences compared to Linux. Should there be any bottlenecks, it would be informative to understand whether these are due to the multi-kernel model or rather specific artifacts of the implementation.
- Barrelfish has a multi-level scheduling infrastructure and there are several interesting topics on operator scheduling which we discuss in Section 5.1. One aspect is whether time slicing on a core could benefit from database knowledge. Another theme worth revisiting is parallelised operators and how they could benefit from coordinated execution.
- Barrelfish has limited tools for debugging and performance analysis. There are a few ideas which come to mind that would make development easier:
  - A simple profiler could be built using the dispatcher upcall and a stack unwinder. The existing tracing infrastructure is useful for short-running programs; this would cater to a need for aggregated statistics from a longer run.

  - The is a family of 'sanitiser' tools [31, 32] which can be used to detect race conditions and memory access errors (e.g. buffer overflow). Unlike more complex tools which rely on dynamic instrumentation these can be easily ported, because they are implemented with compile-time instrumentation with calls to a small runtime library.

  - The system call interface that `libbarrelfish` uses is quite slim. Would it be possible to build a compatibility layer and run Barrelfish applications in a Linux process? This would give access to conventional tools and could be helpful when porting.

- A common application structure on Barrelfish is to place a representative on each core and communicate explicitly using messages. By contrast, SharedDB runs as a single process and makes use of shared memory. What are the implications and design trade-offs if operators instead each run in a separate domain? This is also an approach which could be used to run SharedDB on a non cache-coherent machine and we discuss this in Section 5.2.

- Energy is a valuable commodity. Can the OS – in its role as global resource manager – encourage applications to be more conservative? There are also architectures with heterogeneous cores (such as ARM big.LITTLE). How would operators be deployed onto cores on such a machine and what are the trade-offs when migrating threads?

- Server consolidation is widely used in industry, but this can be problematic for performance-sensitive code. Imagine a co-located application that competes for the shared cache, causing a deterioration in performance. Jails/containers provide containment on a software level. Can techniques such as page coloring be used to provide working sets which don't conflict in cache? A use case for this would be for the streaming query-data join in Crescando.

Appendix A

Appendix

A.1 NUMA awareness

This presents a short summary of what NUMA architectures are and the measures SharedDB implements for such systems. This information complements the experiment described in Section 4.3.2.

NUMA (Non-Uniform Memory Access) describes a particular hardware design where the cost varies for accessing different regions of memory. This can be found in modern multi-core servers and is implemented to improve scalability. Each group of cores (node) is attached to a private area of local memory which can be accessed more cheaply. Processors are still cache-coherent and have a view of the entire shared memory, however accessing remote memory will incur a transfer across the interconnect link. From the software design perspective there are to main aspects to consider: finite bandwidth of the interconnect and latency penalty upon access (typically 50% or more).

`libnuma` [17] is a library which provides information on the hardware topology and allows to allocate memory according to a number of policies. SharedDB makes use of this when deciding where to place an operator and when allocating memory buffers. More concretely here are two scenarios of how this interface is used:

i There are also interfaces to explicitly allocate memory on the local node, on a particular node or interleaved over all nodes. An operator's stack and heap are allocated on the local node to minimise cross-node traffic. NUMA-aware allocation is an explicit operation so quite a large fraction of memory is actually just delivered by the

default allocator.

ii The order in which cores are numbered differs significantly between vendors. SharedDB permutes this to encode its placement heuristic, namely to spread operators across separate chips and thus maximise utilisation of shared resources. The numbering scheme goes sequentially over the cores on a chip, interleaved with each NUMA-domain round-robin. Hyper-threads are placed last to discourage their usage.

A.2 Performance Counters

In this section we give a brief overview of performance counters and the available tools. Consider also reading Section 4.3.4 which discusses how we tried using these to characterise resource usage of operators.

Within the processor hardware there are a fixed number of performance monitoring units (PMUs) which can record a vast number of events. These are very useful for performance analysis because they operate with low overhead and provide insight into micro-architectural details such as requests to the different cache levels or why cycles are being wasted. The processor vendors provide guides [10, 20] with details of the available events and formulae to interpret the raw values.

A piece of software programs a register with a specific event and threshold which arms the counter. Events are gathered and once the desired threshold is reached an interrupt is generated. At this point the program can capture the execution context (instruction pointer, unwind the stack). Later this is aggregated to a report which gives the percentage of events at a given source location. This mode of recording is similar to the operation of a profiler and is referred to as sampling. In some use cases it is more convenient to add explicit instrumentation for which there are libraries such as PAPI [28].

There are a number of different tools available on Linux, for example OProfile, the 'perf events' framework and Intel PCM (Performance Counter Monitor). We have found the first to be flexible and well suited to our needs. In particular, it allows fine-grained control over the events recorded and has several options to customise how reports are generated. Initially we tried the perf framework and while the initial setup is easier we encountered some problems. For instance, it tries to help with platform diversity by providing abstract events however this makes it more challenging to interpret results because the event semantics are

less precise. In our setting we also needed to zoom in on specific cores but we found that events were already pre-aggregated in the trace. The perf framework is undergoing active development and would be worth re-visiting at some later stage. We made use of the last tool for a specific use case which we now briefly describe.

Memory bandwidth and cross-NUMA traffic can be easily measured on AMD systems because there are corresponding events. The same, however, is not true for the Intel Nehalem platform. Such a chip consists of the processing cores themselves and the 'uncore' portion: last-level cache, memory controller and interconnect (QPI). Whilst there is a PMU residing on the uncore, events need to be recorded explicitly over a time duration rather than with sampling. Further, events cannot be attributed to the originating core but rather only at the socket-level. With the more recent Sandy Bridge platform it is now possible to measure memory requests and attribute this on a more fine-grained level [9].

Bibliography

[1] Gustavo Alonso, Donald Kossmann, and Timothy Roscoe. SWissBox: An architecture for data processing appliances. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 32–37, 2011.

[2] Jesper Louis Andersen. How Erlang does scheduling. http://jlouisramblings.blogspot.dk/2013/01/how-erlang-does-scheduling.html, 2013. [Online; accessed 05-May-2013].

[3] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[4] Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Nathan C Burnett, Timothy E Denehy, Thomas J Engle, Haryadi S Gunawi, James A Nugent, and Florentina I Popovici. Transforming policies into mechanisms with Infokernel. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 90–105. ACM, 2003.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[6] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu,

Yuehua Dai, et al. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57. San Diego, CA, 2008.

[7] Scott A Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 396–407. IEEE, 2003.

[8] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 838–849. VLDB Endowment, 2003.

[9] Roman Dementiev. Monitoring integrated memory controller requests in the 2nd, 3rd and 4th generation Intel® Core™ processors. http://software.intel.com/en-us/articles/monitoring-integrated-memory-controller-requests-in-the-2nd-3rd-and-4th-generation-intel, 2013. [Online; accessed 05-May-2013].

[10] Paul J Drongowski. Basic performance measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ processors. *AMD whitepaper*, 25, 2008.

[11] Adam Dunkels. Design and implementation of the lwIP TCP/IP stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[12] Dawson R Engler, M Frans Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 251–266. ACM, 1995.

[13] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. *Operating systems review*, 33:87–100, 1998.

[14] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, February 2012.

[15] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. COD: Database / operating system co-design. In *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2013.

[16] Google Inc. include-what-you-use - a tool for use with Clang to analyze #includes in C and C++ source files. https://code.google.com/p/include-what-you-use/, 2011. [Online; accessed 05-May-2013].

[17] Andi Kleen. A NUMA API for Linux. Technical report, Novell Inc., August 2004.

[18] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42: building a complete operating system. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 133–145. ACM, 2006.

[19] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[20] David Levinthal. Performance analysis guide for Intel® Core™ i7 processor and Intel® Xeon™ 5500 processors. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2008. [Online; accessed 14-March-2013].

[21] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[22] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[23] Jochen Liedtke et al. On-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, 1995.

[24] Brian D Marsh, Michael L Scott, Thomas J LeBlanc, and Evangelos P Markatos. First-class user-level threads. *ACM SIGOPS Operating Systems Review*, 25(5):110–121, 1991.

[25] Daniel A. Menascé. TPC-W: A benchmark for e-commerce. *Internet Computing, IEEE*, 6(3):83–87, May 2002.

[26] Andreas Merkel and Frank Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. *ACM SIGOPS Operating Systems Review*, 42(4):1–12, 2008.

[27] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, pages 153–166. ACM, 2010.

[28] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[29] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX Workshop on Hot Topics on Parallelism (HotPar '10)*, June 2010.

[30] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the sixth conference on Computer systems*, pages 17–30. ACM, 2011.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[32] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitry Vyukov. Dynamic race detection with LLVM compiler. In *Runtime Verification*, pages 110–114. Springer, 2012.

[33] Benoît Sigoure. How long does it take to make a context switch? http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, 2010. [Online; accessed 05-May-2013].

[34] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981.

[35] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, August 2009.

[36] Jianrong Zhang. Characterizing the scalability of Erlang VM on many-core processors. Master's thesis, KTH, School of Information and Communication Technology (ICT), January 2011.