Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 8

Systems Group, Department of Computer Science, ETH Zurich

Database-Operating System Co-Design

by

Jana Giceva

Supervised by

Prof. Dr. Gustavo Alonso

November 2010–May 2011

inf | Informatik
Computer Science

ETH Zurich

Abstract

Systems Group, Department of Computer Science
ETH Zurich

Master Thesis

Database-Operating system co-design

by Jana Giceva

We want to investigate how to improve the information flow between a database and an operating system, aiming for better scheduling and smarter resource management. We are interested in identifying the potential optimizations that can be achieved with a better interaction between a database engine and the underlying operating system, especially by allowing the application to get more control over scheduling and memory management decisions. Therefore, we explored some of the issues that arise in database-operating system co-design.

Particularly, during the course of the thesis, we ported an existing, main memory, column store engine on top of the Barrelfish OS. Both are research systems developed in the Systems Group at ETHZ addressing the main issues of scalability, hardware heterogeneity and application constraints and requirements. Our extensive analysis shows that the performance of the resulting system and its behavior are stable and resemble the baseline of the columnstore engine's run on top of Linux. Furthermore, the experiments also show that the scale-up version implementation scales up almost linearly with the number of cores. We conclude this report with a discussion that even though currently the column store engine is too simple to be able to fully utilize the scheduling capabilities and resource management features provided by Barrelfish, the two systems form a solid foundation that can be enhanced in the future with a set of more complex features. This will eventually result in a fully functional database engine tightly collaborating with the OS via well defined interfaces.

Acknowledgements

I would like to thank my mentor Prof. Dr. Gustavo Alonso for his invaluable support and direct guidance during the course of this thesis.

I would also like to thank Prof. Dr. Timothy Roscoe for the great discussions and useful advice and suggestions.

Furthermore, special thanks to Simon Peter and Tudor Salomie for helping me with the Barrelfish OS and the Shared Scans on Column Stores (CSCS engine). I would also like to thank Akhilesh Singhania, Pravin Shinde, the rest of the Barrelfish team, and my friends and family for their support during the thesis.

Contents

Abstract i

Acknowledgements ii

List of Figures v

List of Tables vi

List of Listings vii

1 Introduction 1
- 1.1 Background and motivation . . . 1
- 1.2 Problem statement . . . 3
- 1.3 Thesis contribution . . . 4
- 1.4 Thesis outline . . . 4

2 Fundamentals 6
- 2.1 Shared Scans and Column Stores . . . 6
  - 2.1.1 Column stores . . . 6
  - 2.1.2 Shared scans . . . 7
  - 2.1.3 Combining Shared Scans and Column Stores . . . 8
- 2.2 The Barrelfish OS . . . 10
- 2.3 Running the CSCS engine on Barrelfish . . . 12

3 Porting the CSCS engine on Barrelfish 14
- 3.1 Challenges while porting . . . 14
- 3.2 Dependencies . . . 15
- 3.3 Modifications on the CSCS engine . . . 15
- 3.4 Modifications on Barrelfish . . . 17
- 3.5 Scale-up design decision . . . 18
  - 3.5.1 Multiple threads reading from the NFS . . . 18
  - 3.5.2 Multiple processes sharing memory area . . . 20
  - 3.5.3 Multiprocess design alternative . . . 22

4 Experiments and Results 24
- 4.1 The workload . . . 24

4.2 Architecture and specifications of the machines used 25
4.3 Methods and tools for performance analysis 26
4.4 Baseline performance 27
4.5 Linux vs. Barrelfish performance analysis 32
4.6 Performance on multiple cores 36
4.7 Scale-up analysis 39
4.8 NUMA support 41
4.9 NUMA effect analysis 42

5 Conclusion **49**
5.1 Thesis contribution 49
5.2 Future work 50

A CMake to Hake translation **52**

B Changes performed on the CSCS engine **55**

C List of all performance counter events used **59**

Bibliography **65**

List of Figures

3.1 Multithreaded version: threads load and process the datastore individually . . . 19
3.2 Multi process version with shared memory segment: processes sharing access to a memory chunk containing the datastore objects 21
3.3 Multi-process version with barrier service for inter-process synchronization 23

4.1 AMD Santa Rosa architecture 25
4.2 AMD Shanghai architecture 26
4.3 Baseline performance: throughput of real Amadeus workload 28
4.4 Baseline performance: response time of real Amadeus workload 29
4.5 Impact of varying the number of updates on throughput 30
4.6 Impact of varying the datastore size on throughput 32
4.7 CSCS on Linux: CPU usage breakdown 36
4.8 Execution time breakdown 37
4.9 Scale-up performance: throughput of real Amadeus workload on AMD Shanghai 38
4.10 Scale-up performance comparison Linux vs. Barrelfish on AMD Shanghai 39
4.11 AMD Shanghai NUMA architecture 42
4.12 Effect of NUMA awareness: DRAM utilization 44
4.13 Effect of NUMA awareness: local to remote NUMA node access ratio 45
4.14 Effect of NUMA awareness: local to other NUMA nodes access ratio 46
4.15 Effect of NUMA awareness on system performance 47

List of Tables

4.1 Linux vs. Barrelfish, performance analysis 33
4.2 STREAM benchmark results 34
4.3 Bandwidth bytes transferred 35
4.4 Scale up - performance analysis 40
4.5 Events used in NUMA analysis 43

C.1 List of AMD performance counter events 59

Listings

A.1 Configuring the values for the Configure.h.in file 52
A.2 Copying files and creating symbolic link to other directories 52
A.3 Creating a library in `CMake` 53
A.4 Creating a library in `Hake` 53
A.5 Creating an executable in `CMake` 54
A.6 Creating an executable in `Hake` 54
B.1 Acquiring duration of execution in Linux 55
B.2 Quiring duration of execution in Barrelfish 55
B.3 Working with threads in Linux 56
B.4 Working with threads in Barrelfish 56
B.5 Handling NFS files in Barrelfish 56
B.6 Spawning multiple instances of the same program in Barrelfish 57
B.7 Setting up process affinity in Linux 57
B.8 Setting up process affinity in Barrelfish 57
B.9 Performance measurements on Linux 57
B.10 Performance measurements in Barrelfish 58

1

Introduction

It is a well acknowledged fact that databases should have more control over the system and the available resources. They are characterized as applications with specific memory and I/O access patterns, and well-defined scheduling requirements. Therefore they can do efficient resource management provided that the OS does not stand in the way by trying to guess its needs and desires. In particular, the OS and the database should cooperate and address the challenges imposed by the hardware trends and application constraints and requirements. This opens up a possibility for an interesting research topic: database-os co-design. In this thesis we do the initial steps in that direction, by porting the CSCS engine on top of Barrelfish and characterizing the performance of the resulting system and showing that they can be used as building blocks for the DB-OS co-design.

We start by giving a brief overview of the thesis motivations. Then we define the problem statement, and we continue with a short summary of the thesis contribution. We finalize with an outline of the subsequent chapters.

1.1 Background and motivation

We classify the main motivation points into three categories: diversity in hardware resources, factors influencing the database research and the option for cross-layer optimizations emphasized in database appliances. In this section, we briefly describe each of these categories.

Diversity in hardware resources

It has been argued in the systems community that diversity of hardware resources will be as much of a problem as raw scalability; as increasingly complex many-core processors become

the norm. In [1], for instance, the researchers have identified three different types of diversity: *non-uniformity*, *core diversity* and *system diversity*. Apart from non-uniformity, which has been well acknowledged by many to have impact on performance, the other two types of diversity have received little attention, although it is almost certain that they will become more and more important in the future. **Core diversity** refers not only to the expected heterogeneity of cores within a single system, but also to the emerging practice of using GPUs and FPGAs for general purpose processing. **System diversity**, on the other hand, refers to the fact that various systems now and in the future will be characterized by a different set of available resources. As a consequence, it would be impractical to write code optimized for each specific hardware.

It is of key importance that a novel operating system that is designed to address the new trends in hardware, takes all these things into consideration and incorporates the proposed solutions as one of their core design principles.

Factors influencing the current research in databases

In recent years, a large number of new techniques and database architectures have been proposed. This has been motivated by several different factors:

- new **hardware trends**: multicore machines that impose scalability problems to traditional DBMSs[2]
- heavy **workloads**: increasing need for support of more complex queries, OnLine Analytical Processing (OLAP), and Operational Business Intelligence (Operational BI) workloads. These workloads are characterized by complex multi-dimensional analytical and ad-hoc queries that require full table scans.
- new **application requirements and constraints**: more demand for deterministic and predictable performance under a range of query complexities and update loads, with latency constraints. Additionally, customer applications want to be able to support a large number of clients at peak loads by scaling on a vast number of processing nodes and at the same time to have a guaranteed reliability for continuous operation (for example [3]). As it was shown in *Crescando* [4] the combination of short response time constraints and at the same time support for a wide variety of loads cannot be handled by existing engines.

Architecture for developing database appliances

SwissBox [5] elaborates on the emerging trend of database appliances. In short, a database appliance is one single system that incorporates all systems needed for data processing solutions, optimizing the design; as well as operational and maintenance costs. An appliance does that

by using specially customized hardware, a specifically tailored operating system that meets the needs of the application and highly optimized software for the underlying platform. SwissBox is an architecture for developing database appliances, developed in the Systems Group. With this architecture the authors target one broad goal: engineering new appliances in the face of diverse and rapidly changing hardware and workloads, the issues we have discussed previously in this chapter.

Thesis motivation

Database appliances also present the idea for cross-layer optimizations for performance and scalability. This can be especially interesting if done between the database and the operating system, where one can investigate how to improve the information flow between the two layers aiming for better scheduling and resource management.

The main idea is to take a database that addresses the workload and scalability issues and incorporate it on a modern operating system that deals with hardware scalability, and future architecture heterogeneity problems. Furthermore, the operating system should also be designed to collaborate with the applications running on top of it.

1.2 Problem statement

This thesis explores the possibility for starting an interesting research direction - *database-operating system co-design* with two systems selected as building blocks: the **Barrelfish OS** and the **CSCS engine**. Both systems are developed in the Systems Group, and a short overview for them can be found in the second chapter of the thesis report.

On a more technical note, some of the questions that we were aiming to answer are the following:

- Is the Barrelfish OS mature enough to port the CSCS engine on top of it? What are the challenges for performing that task? What modifications are needed on both sides, so that the two systems can work together? Can it serve as a solid base on which we can build more complex applications such as a database engine?
- What is the performance of the CSCS engine running on top of Barrelfish? How does it scale, both with increased workload and dataset size on which it operates? What are the system performance hot-spots? Are there bottlenecks that we can further optimize and if so, what are the available options?
- How can we extend the CSCS engine to scale-up to multiple cores on Barrelfish?

- How do we compare the nature and characteristics of the system's behavior to a more traditional setting, when the CSCS executes on Linux? What are the observed differences? Can we explain them? Can we identify the factors that influence it the most?
- How does the new compound system behave on different hardware architectures?
- How can we extend the work done so far? Ideas for short- and long-term future work.

1.3 Thesis contribution

The main contribution of the thesis is the achievement of making two research systems, the CSCS engine and the Barrelfish OS, to work together. Their joined performance and behavior resemble the baseline of the CSCS engine's run on Linux. A more detailed instrumentation analysis confirmed that, apart from offering the desired basic concepts, Barrelfish and the CSCS engine form a solid system foundation that can be enhanced with more complex features; eventually resulting in a fully functional database engine tightly collaborating with the OS via a well defined interface.

The main steps that summarize the work done during the course of the thesis are the following: We first ported the CSCS engine and its dependencies on top of Barrelfish. While porting, various changes were made on both systems, process that is explained in more details in Chapter 3. Then we continued with experiments that compare the performance of the new system with a more traditional variant (CSCS engine executing on top of Linux). Both executions were further analyzed by gathering a range of performance events that helped us profile and characterize both systems and compare them in a more fine grained level. Afterwards, we extended the execution of the program to scale-up on multiple cores on Barrelfish. For this purpose, several design alternatives were carefully considered. This was also followed by a number of experiments that excersized the system's scalability properties. We then continued with performance analysis when running on multiple cores. Last but not least, the compound system's performance tests were repeated on two different architectures. The results obtained from all experiments can be found in Chapter 4.

1.4 Thesis outline

The rest of the thesis report is organized as follows:

- **Chapter 2** gives a background and a short overview of the underlying systems: CSCS engine and Barrelfish OS.

- **Chapter 3** elaborates on the work that has been done to make the two systems work together, the challenges encountered and the modifications made on both systems.
- **Chapter 4** presents the results of all experiments done and the performance analysis performed on the resulting system.
- Finally, the conclusion and future work ideas are covered in **Chapter 5.**

2

Fundamentals

This section gives a short overview of the main systems that were used as building blocks for the database-OS compound system: the CSCS engine and the Barrelfish operating system.

2.1 Shared Scans and Column Stores

In order to better understand the Shared Scans on Column stores a short introduction of what column stores and shared scans are follows.

2.1.1 Column stores

Traditional database management systems (DBMSs) use a row store as their storage layout. Consequently, whenever it needs to evaluate a tuple of the table it has to load and process the whole row, no matter the size, even though sometimes the query only requires evaluating the values of one specific column.

A column store, as the name suggests, stores a relational table by columns i.e. for each column all values of it are serialized together. In the past decade, column-based DBMSs became an important alternative to existing row-based relational DBMSs, attracting a lot of attention both in the research communities and in the industry. Column stores are particularly attractive for *OLAP* and *Operational BI* workloads because they involve processing a large number of tuples in order to compute a certain business metric. If the query involves only a few columns of a table, then only a fraction of the data will be processed if a column store database engine is used.

Today there are quite a few column-oriented DBMS implementations, for example: disk oriented *Vertica* and *C-Store*[6], as well as in-memory databases such as *MonetDB*[7] and SAP's *T-Rex* accelerator. They all claim to be faster than traditional DBMSs for read intensive workloads.

2.1.2 Shared scans

Multi-query optimization has been around for almost thirty years[8]. Its simple instantiation are the shared scanes. Rather than searching for any kind of common sub-expression, this technique only shares the most expensive and at the same time the simplest operation - the scan.

Like column stores, shared scans are also attractive for *OLAP* and *Operational BI* workloads. The key idea is to group queries that operate on the same table and execute the scan on that table only once for all queries of the batch.

Shared scans have been adopted by a number of database systems including *RedBrick*[9], *IBM Blink*[10], and *Crescando*[4].

For main memory databases, shared scans have been studied in the context of row stores. Crescando for example proposes a specific shared scan operator called *ClockScan*. One of the key ideas behind Crescando's ClockScan is to index the batch of queries over the query predicates. That results in better performance - being able to process and answer thousands of queries with one shared scan.

Crescando's ClockScan

The goal of Crescando was to design a technology that is able to answer a large number of unpredictable queries and updates within a predictable time for a relational table. For comparison the traditional systems were designed to primarily optimize the performance for the most common queries, by building indexes and adding views.

Each scanning thread of Crescando is a kernel thread with a hard processor affinity, that continuously scans a horizontal partition of the data, residing in main memory, and outputs the resulting tuples. A scan thread can handle thousands of requests (both queries and updates) at a time.

As mentioned earlier, the scanning thread of Crescando uses the ClockScan algorithm. It iterates over all records of its data partition and performs a query-/update-data join over a set of operations. In a query-data join the queries are looked at as a relation of predicates. The query-data join is implemented as an index union join, where an index is built over the query predicates, in contrast to traditional database systems, where the data in table is indexed.

In the end, every query belongs to one of two sets:

- indexed queries: those that have a predicate that is part of the index
- unindexed queries: all the rest

Briefly, after indexing the queries, the scanning thread does the following work:

Data: DataChunk c, IndexSet is, QuerySet qs
Result: ResultSet rs

```
foreach Record r ∈ c do
    //check if there is an index for the current record
    foreach Index i ∈ is do
        QuerySet C ← Probe(i,r);
        foreach Query q ∈ C do
            rs ← Execute(q,r);
        end
    end
    //execute unindexed queries
    foreach Query q ∈ qs do
        rs ← Execute(q,r);
    end
end
```

ALGORITHM 1: ClockScan

This algorithm simply iterates over all the records belonging to the given data chunk, and evaluates each record first against the index of queries (provided that there is a non-empty set of queries forming an index over that record) and then probes it over all the other unindexed queries.

The algorithm of ClockScan has been showed to be CPU bound. For more information about the ClockScan and Crescando in general please refer to [4]

2.1.3 Combining Shared Scans and Column Stores

The system that we have chosen to use - CSCS engine, explores how to implement shared scans on column stores [11] Experiments have shown that shared scans combined with column store layout achieve better performance than any approach exploiting only one of these techniques.

The challenge the authors see is because it needs to process many requests over data partitioned into columns. This requires performing a three-way join: requests (i.e. queries or updates), columns (i.e. attributes) and tuples (i.e. rows). Additionally, the implementation has to make sure to optimize for cache utilization and to maximize locality.

The scanning thread executes the following algorithm:

Data: QuerySet qs, IndexSet is, Predicates p
Result: ResultSet rs

```
ResetPositionList(plist);
predAttr ← GetPreds(qs).Sort(p);
foreach Column col ∈ (is ∩ predAttr) do
    col.PopulateIndex(q);
    foreach Record val ∈ col do
        plist ← col.Probe(val);
    end
end
foreach Column col ∈ (predAttr − is) do
    activeQueries ← GetQActiveOnCol(qs,col);
    foreach Query q ∈ activeQueries do
        foreach Tuple tuple ∈ plist[q] do
            plist ← col.EvalPredicate(q,tuple);
        end
    end
end
foreach Query q ∈ qs do
    rs += GetResultTuples(plist[q]);
end
```

Algorithm 2: ColumnStore ClockScan

The algorithm starts by resetting the values in the position list. It then determines the set of columns used as predicates in the batch of input queries, and sorts them based on their selectivities. The first two nested for loops perform the following task:
Iterate over the columns that belong to the input set of "columns to be indexed". For each of these columns create a query index, that is populated using the corresponding predicate values of the queries. The inner loop probes each value in the column against the query index, and stores the matching tuples (query and tuple identifiers) in the position lists.
The next three nested loops perform the following task:
Iterate over the remaining columns and extract a subset of the original batch of queries that has a predicate on the current column. Furthermore each of them must not be marked in the first half of the algorithm that it does not have a match on this particular column. Then, for each query in this set, test its predicate on the current column and then update the position list. In the last for loop, iterate over the batch of queries and generate the result tuples using the information stored in the position list.

Updates are handled in a similar way to queries. The only restriction is that all the queries need to see the updates belonging to the same batch. Consequently, there exists a second scanning thread, running in parallel, performing the updates. Another constraint is that we need to make sure that the updates are executed in same order (total order) in which they arrived in the system (update input queue).

For brevity, the algorithm used is not included in this short summary. More details about the system can be found at [11]

2.2 The Barrelfish OS

The Barrelfish OS is a research operating system designed to manage heterogeneous and multicore environments. It is an implementation of a new OS structure - the "multikernel"[12]. Multikernel treats the machine as a network of independent cores, which communicate exclusively via message passing. Furthermore there is no inter-core sharing at the lowest level, rather replication and partitioning. That makes this new OS structure not only a better match for the underlying hardware, but also one that allows us to apply insights of distributed systems to the problems of scale, adaptivity and diversity in operating systems for future hardware.

Handling hardware heterogeneity

Traditional operating systems hide heterogeneity by abstracting the underlying hardware. Therefore system components and applications see a homogeneous SMP system where every core appears the same. This no longer mirrors the current developments in the hardware technology. As a result, by hiding the underlying heterogeneity, the applications have no chance to optimize their execution on the appropriate hardware components.

Most of the existing work to date on dealing with hardware diversity in commodity operating systems has been concentrated on the performance issues in NUMA architectures. However there has been relatively little work dealing with core and system diversity. Similar to [13][14], Barrelfish minimizes the resource allocation policy in the kernel and delegates management of resources to application as much as possible. At the same time it enables the subsystems and the applications to make better use of the hardware capabilities and to deal with heterogeneity as best as possible.

Barrelfish combines a rich representation of hardware in a subset of first-order logic and a powerful reasoning engine, in a single system knowledge base (**SKB**)[1] service. It is populated by hardware discovery information (e.g. interconnect topology, cache parameters, etc) as well as online measurement of the utilization of the resources. This way, applications can query the

SKB service to get a more detailed description of the raw resources for the intra-application resource management.

Barrelfish scheduling

Due to the distributed nature of Barrelfish, the **scheduling** is implemented at multiple timescales: long, medium and short term. In addition to the possibility to reason online about the hardware characteristics and utilization, an application should also expose as much as possible information about its current workload and resource requirements to help the OS effectively allocate the resources. In Barrelfish this can be achieved with the *scheduling manifest*, which is a specification of each application's predicted long-term resource requirements. This is used together with the current hardware utilization to determine a suitable set of resources for long-term allocation to that particular application. Furthermore efficient overall operation requires continuous two-way information flow between the applications and the operating system. Applications should indicate their ability to use or release resources, and the OS should signal an application whenever new resources have been allocated to or preempted from it. In Barrelfish, the applications express short- and medium-term scheduling requirements to the OS by placing dispatchers into one or more *dispatcher groups* and then negotiating how each group is to be scheduled.

For more details on the exact implementation please consult the Barrelfish scheduling paper[15].

Brief technical introduction

On a more technical side, in this subsection we briefly introduce some concepts and notions that are of relevance for the thesis.

A **process** in Barrelfish is represented by a collection of *dispatcher* objects, one on each core on which it might execute. Similar to [16] dispatchers on a core are scheduled by the local CPU driver, which invokes an upcall interface that is provided by each dispatcher. Above this upcall interface, a dispatcher runs a core-local user-level thread scheduler. The schedulers on each dispatcher exchange messages to create and migrate threads between each other and hence between cores.

Even though it is based on shared-nothing principles Barrelfish does offer the notion of **shared address space** for its applications. The virtual shared address space can be achieved either by sharing a hardware page table among all the dispatchers belonging to one process, or by replicating the hardware pages.

Resource management in Barrelfish is handled with the help of *capabilities*. Briefly, a **capability** is an opaque, unforgeable token which refers to a region in memory. Possessing it confers

the right to invoke (via a system call) operations on the object residing in that region. In this sense, the resource management in Barrelfish can be summarized as follows:

1. mediating the transfer of capabilities (and thus resources) between entities (and thus cores);
2. ensuring that these resources are used correctly by enforcing a set of rules;
3. enforcing revocation of resource access whenever a capability is revoked;

For more detailed explanation on how capabilities are implemented and transferred between entities, and cores please consult [17].

In a multikernel, all **inter-core communication** occurs with messages. All message transports are abstracted behind a common interface, allowing messages to be marshaled, sent and received in a transport-independent way. *Flounder* is a stub compiler that compiles IDL specifications into code for (un)marshalling messages: server and client stub. The message `send` function by default is non-blocking, and hence returns immediately with a success/failure depending whether the message sent has been enqueued for further transfer or not. This message-based asynchronous communication serves as a solid foundation for having inter-core services, that can be created, published and invoked by other processes via message communication.

For more detailed information regarding the implementation and performance evaluation of the inter-core communication in Barrelfish please refer to [18]

2.3 Running the CSCS engine on Barrelfish

On one hand we have an engine that successfully responds to the increasingly popular OLAP and Operational BI workloads and scales well with the underlying multicore hardware, at the same time having predictable performance: high throughput and a guaranteed low response time. On the other hand we have an operating system that is not only designed for heterogeneous and multicore future hardware but also embraces the need for OS-application collaboration and provides mechanisms for more fine-grained scheduling and efficient resource management.

This opens up a possibility for OS-database co-design: architecting both systems at the same time, by creating appropriate interfaces between them. With it we can finally address the always persisting tension between operating systems and DBMSs over resource management. We can explore how to pass the database knowledge of the read/write patterns and internal thread organization to the OS so that the OS can make intelligent decisions rather than getting on the way or having to guess the application needs. These points were also mentioned in [5]

To start with, we have to first investigate how these two systems can be made to work with each other, and what the performance characteristics of the resulting system are.

The following two chapters (3, 4) elaborate more on the work done for porting the CSCS engine on top of Barrelfish and the experiments conducted. Chapter 4 also concentrates on a more detailed analysis of the performance of the resulting system, eventually concluding that it is a solid foundation for future work in the direction of *database-os co-design*.

3

Porting the CSCS engine on Barrelfish

The CSCS engine was developed as a toolkit for testing Column Store design and multiquery optimizations. The best version for a given workload was then ported to the Barrelfish OS.

3.1 Challenges while porting

In general the challenges when porting the CSCS engine were due to several differences between Barrelfish and Linux, on which the database was initially implemented. A short overview of the important differences between the systems, as seen from an application that is to be compiled and executed on top of it:

- kernel: standard Linux kernel vs. the multikernel implementation in Barrelfish
- `C` library: the standard GNU `libc` library[19] was used in Linux, and a new `libc` implementation developed by NICTA [20] and the Barrelfish team, was available on the Barrelfish system.
- `C++` library: the standard GNU `libstdc++` [21] was used on the Linux system, and a newly developed `libc++` [22] library, targeting the `C++0X` together with the `libsupc++` were used in Barrelfish.
- GCC compiler versions: GCC 4.4.5 on Linux, and GCC 4.3.5 on Barrelfish

One of the main drawbacks for the `C++` support on Barrelfish, is the lack of support for exceptions. Even though that was not a direct issue with the ClockScan on Column Stores

implementation, it was a blocker for porting some of its dependencies, as shortly discussed in the following subsection.

Another not-so-trivial challenge was to translate the CMake [23] content, used to build the database on Linux, to the language of the Barrelfish building tool, Hake [24]. More information and guidelines on general rules how to translate a regular CMake to Hake can be found in Appendix A.

3.2 Dependencies

Before porting the ClockScan on Column Stores database on top of Barrelfish, we had to port its dependencies: the **Boost Library** [25] and **Google Hash library** [26]. Both of them are implemented using `C++`, and the modifications necessary were just some minor changes regarding naming and locating some of the standard `C/C++` header files and their implementations. Furthermore, it was necessary to make a clear separation between the `C` and `C++` header files using the `extern "C"` block. The only major blocker, especially for the Boost Library, was the lack of support for the `C++` exceptions, on which it is heavily dependent. For that purpose, only the necessary modules of the Boost library were ported and tested, and the rest is left for future work after the exceptions support is made available.

3.3 Modifications on the CSCS engine

This subsection presents some of the modifications that were performed on the CSCS engine.

Thread management and synchronization primitives

Barrelfish has its own implementation of multithreading functions and synchronization primitives, and in general it has the same available functionality as the standard POSIX threads, with similar API. Some of the specific characteristics noticed were the following:

- There is support for a multithreaded application to run on a single core, but with certain limitations. Only the main thread is allowed to control the resources given to the application, as well as to allocate/deallocate memory, interact with the VFS and the other cores. The other "secondary" threads are only able to perform computations on the already allocated resources without modifying anything. It was therefore important to make sure that the query and update execution threads are restricted to respect these limitations without affecting both their execution correctness and performance.

- Differences in the implementation of lock and unlock functionality. Linux allows a thread to unlock a specific critical section more times than it was locked. In this sense it allows for less rigorous control in the implementation of a multithreaded program. This however is not the case in Barrelfish. A certain critical section can be unlocked exactly the number of times it was locked, any additional call to unlock will result in an error message printed on the screen. This required to rearrange some of the locking mechanisms so that it follows the constraints imposed by Barrelfish.

For more detailed and technical documentation on the exact changes performed on the system please consult Appendix B

Reading from and writing to VFS/NFS

Reading from and writing to bigger files, requires that we include them by mounting an NFS directory, where they reside. A good example for this is the data-store, whose size could reach up to 7GB. The changes required to the CSCS engine were minor: initializing the VFS service, mounting the desired folder in a specific directory and then using the path to that particular directory in the future data-store accesses. More detailed documentation can be found in Appendix B

Usage of performance counters for instrumentation

In Barrelfish there are no advanced tools for doing the instrumentation external to the program, such as for example OProfile [29]. Instead we had to use the provided functionality for configuring and reading from the available performance counter registers. This enabled us to get more fine-grained granularity of the events, only for the portion of the program of our interest (the scanning threads). The exact code excerpts of the performance counter support API used can also be found in Appendix B.

Spawning processes or threads from within a program

As discussed earlier in Chapter 2, a process structure in Barrelfish is represented by a collection of dispatchers. The dispatchers can share a domain and together with a core-local user-level scheduler manage the threads belonging to the same process. Having this characteristic, Barrelfish introduces a new set of API functions that are at our disposal for implementing the scale-up version of the CSCS engine. More detailed description and analysis of the investigated designs will be provided in one of the following subsections.

In order to spawn the programs and applications to run when booting Barrelfish, one needs to include the appropriate binaries from within a file (`menu.lst`). For each binary, we specify the location where it can be loaded from, the core on which it should run and the program arguments. However, whenever we want to spawn the same applications on multiple cores, it makes little sense to include the same binary several times, especially when its size is relatively big. Barrelfish offers functions that allow spawning the same process on another core with different input arguments, from within the process itself.

3.4 Modifications on Barrelfish

This subsection presents some of the problems that we detected in Barrelfish, when porting the database on top it.

Memory management implementation issues

We detected a limitation imposed by Barrelfish in the memory management implementation. Whenever an application requested to allocate a memory chunk of certain size, the function accepting the client's request asked the memory server for a capability (see Section 2.2) of that size. The problem occurred when the memory server was not able to find such a capability, although it had available memory, due to memory fragmentation. This resulted with an error sent back to the user. With the new fix, the function accepting the application's request will keep on retrying to get a capability of that size by asking the memory server for several smaller-sized capabilities until it either succeeds or fails with the lowest possible page size. If the latter happens it returns with `out-of-memory` fault. The fix was extended to also check for a malloc memory overflow, and with a proper error call stack hierarchy.

Network stack driver issues

Networking stack driver was found to have several issues:

- There was no support for multiple concurrent applications, running on different cores, to be able to mount the NFS at the same time. During the course of the thesis a new networking stack driver was implemented and was later used when running the multicore scale-up experiments.
- The networking driver does not have support to handle multiple connections to the e1000 driver and distinguish whether the upcoming connection is coming from an application

that already has established a connection, but from another dispatcher/thread or from another application.

- The data transfer is based solely on exchanging RPC messages and is therefore slow. This introduces a considerable time overhead when loading a data-store of large size. At the moment of writing the thesis there is ongoing work on implementing a bulk transfer that should solve the data-transfer speed problem.

Performance Counter support

The existing support for performance counters limits us on the AMD machines and on using only one of the available registers. This was extended so that it utilizes the second register as well. Future work would involve further expanding it to support all possibly available registers and to have the identical support also available for the Intel machines.

3.5 Scale-up design decision

Once the program was ported on top of Barrelfish, running on one core, we were faced with another task of extending it to exploit multiple cores and stress the usage of more resources. For this purpose we considered several design alternatives which are presented in the following subsections.

3.5.1 Multiple threads reading from the NFS

The initial idea was to make the application process spawn several identical threads that will perform the same functionality as the main CSCS process until now. These threads will run on different cores, but will belong to the same domain.

Background

As we have explained before a process in Barrelfish (see Section 2.2) may consist of a set of dispatchers, one on each core, that handle some of the process' threads. These threads share a virtual address space, and are scheduled by a user-level thread scheduler running on top of each dispatchers.

We have also mentioned that apart from sharing the address space, most of the applications need the ability to transfer capabilities across cores. In Barrelfish, monitors provide the support for sending capabilities from one core to another. (see Section 2.2)

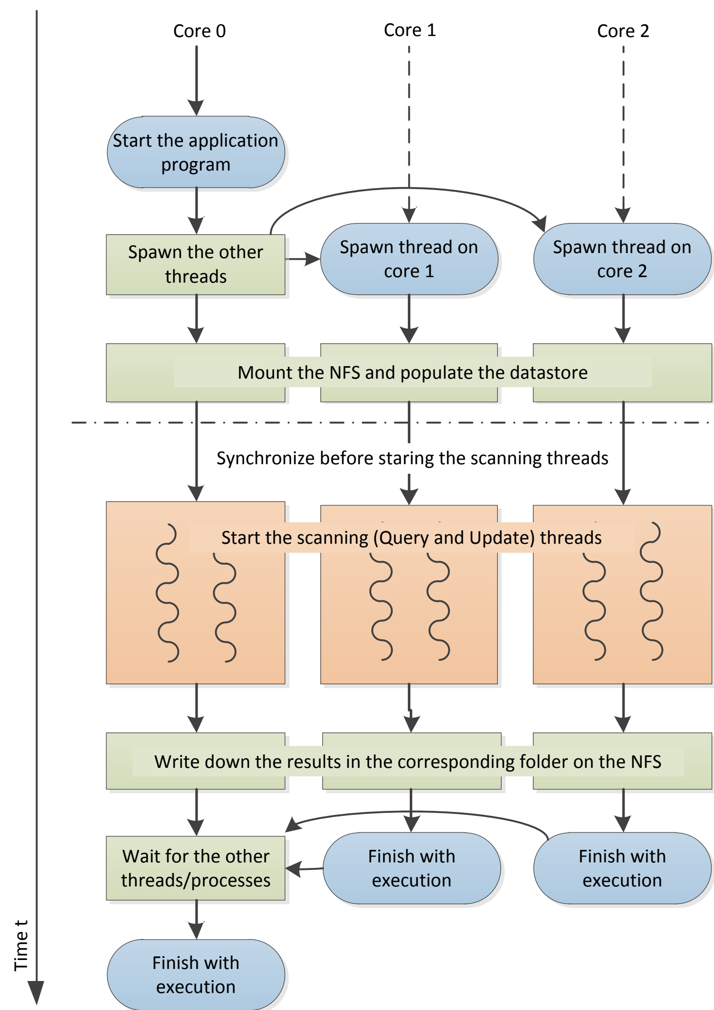


FIGURE 3.1: Design alternative 1: multiple threads loading and processing the datastore individually

Design

As we can see from figure 3.1 the main parent thread spawns several threads to run on different cores. It then mounts the NFS, gets the capability for reading/writing to VFS and sends the capability to all other threads. They will then mount the data-store and create additional two threads running on their local cores that would perform the query/update ClockScans in the system and write out the obtained results and measured statistics back to the corresponding NFS file/folder.

Problems and Limitations for the proposed design

The problem occurs when the threads running on the other cores try to access files residing on the VFS, because at the moment there is no support for using the VFS simultaneously across multiple dispatchers.

Even if we transfer the capability from the core on which the NFS was initially mounted to all the other cores it will not work, as it is a capability only valid for a local connection.

3.5.2 Multiple processes sharing memory area

Another idea would be to have one process that will mount the NFS, and do all the initialization and population of the columnstore data-structures. These data-structures reside in an inter-process shared memory chunk. The main process will then spawn other processes that will operate on the already populated columnstore data-structures.

Background and Requirements

On one side we had the limitation on reading from the VFS from multiple dispatchers so we can not consider a multithreaded variant. On the other side, our network stack driver was still not stable to support multiple applications accessing it at the same time. This lead us to the design of having multiple processes that will share a big memory chunk. The main process will mount the NFS, create the shared memory area, populate it with datastructures containing the information from the NFS and then share the capabilities to access this memory area with the other processes.

As presented earlier (Section 2.2), Barrelfish provides support for creating services that can be accessed from within other processes via message passing. Having this in mind, we have implemented a service that when asked, will create a shared memory area given the size, get a capability for it and return it to the application requesting it. It will then continue listening for, and accepting requests from other processes. These processes want to get the capability for the same shared memory area. The service will respond them with a pointer to the shared memory area and the capability for accessing it.

Design

As we can see from figure 3.2 the main process mounts the NFS and then requests from our external service a capability for accessing a memory region of specific size. It then reads the binary data files from the NFS and populates the data-store datastructure residing in the shared

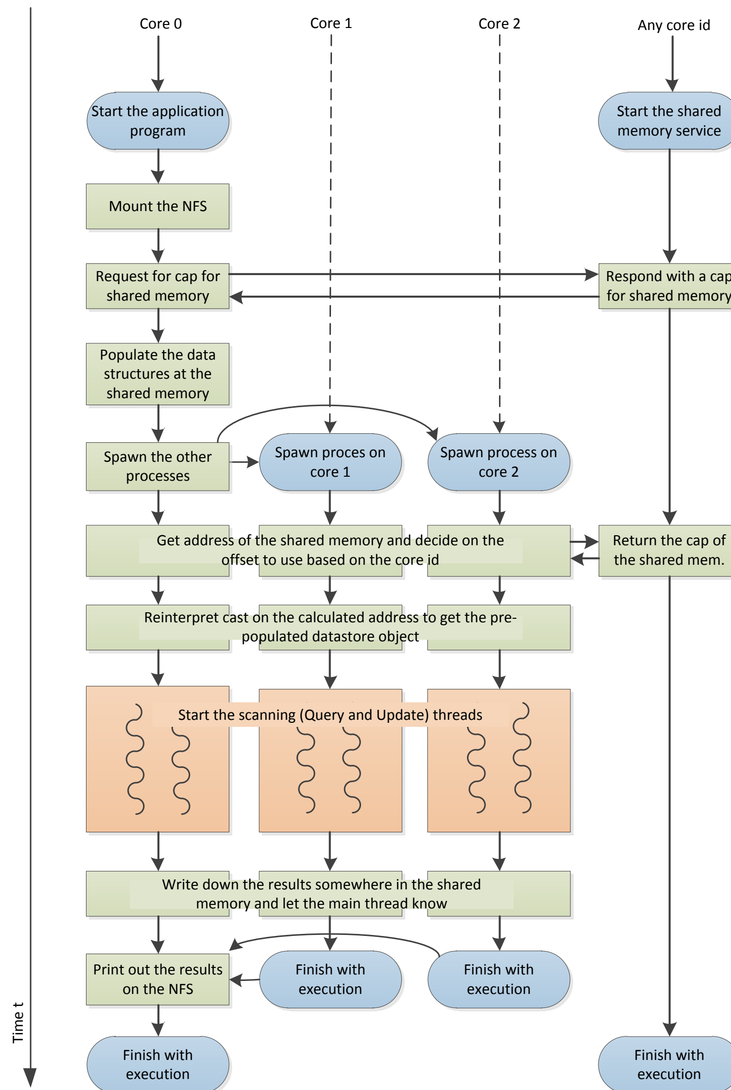


FIGURE 3.2: Design alternative 2: multiple processes sharing access to a memory segment, containing the populated datastore objects

memory region. The other processes are spawned once the data-store objects are populated. They invoke the same external service to get the required capability for reading from the shared memory area. Once they receive the capability, each of them calculates the appropriate offset based on their core id. Then they reinterpret cast the object residing at the calculated offset. Once they have the data-store object they start the scanning (Query and Update) threads, and eventually write down the obtained results in the same shared memory area, exit and let the main process know they are done. Once the main process receives the signal that everyone is done with execution, it outputs the results on the NFS and finishes its execution.

Problems and Limitations for the proposed design

There are several problems with this design. On one hand this solution does not scale well with the number of cores available on a machine. Especially if the underlying architecture has NUMA characteristics. The shared memory will reside on one or a few of the NUMA nodes, and this would, in some cases, introduce a high overhead when accessing the data-store from a core belonging to a node several hops away. Furthermore it increases the traffic on the NorthBridge and HyperTransport^{TM} Links. On the other hand, the data-structures that are used to store the content of the data-store are objects that contain many pointers. Reinterpret-casting these objects in another process, running in a completely different address space would introduce a lot of pointer mess. Last but not least, using the shared memory to copy the binary files instead of directly populating the data-store datastructure, will not scale as the size of the binary files containing the data-store information is too large to fit in a regular scenario.

3.5.3 Multiprocess design alternative

This left us with the only possible alternative that required fixing the network stack driver to allow us to have multiple applications mounting and reading from the NFS.

In this alternative the same application is spawned on different cores; and every instance mounts and reads from the NFS. The problem that occurs is that the applications finish reading from the VFS at different times (within a range of a few minutes). Consequently they do not start processing the queries and updates at the same time, so we had to find a way to synchronize them.

Background and Requirements

As mentioned in the previous design alternative, we can rely on an external service, the barrier, for synchronizing the runs of the separate processes. This service is started as a separate process. It accepts requests from other processes that ask for permission to continue with their execution. The requests are blocked until the barrier service responds with a message that releases all of them at once. The service unblocks the processes once it receives a certain number of requests (the number is given as initial parameter when spawning the barrier process).

Design

As we can see from figure 3.3, the main process spawns the other processes, identical to it, to run on other cores. Once they have populated the data-store and are ready to start with the

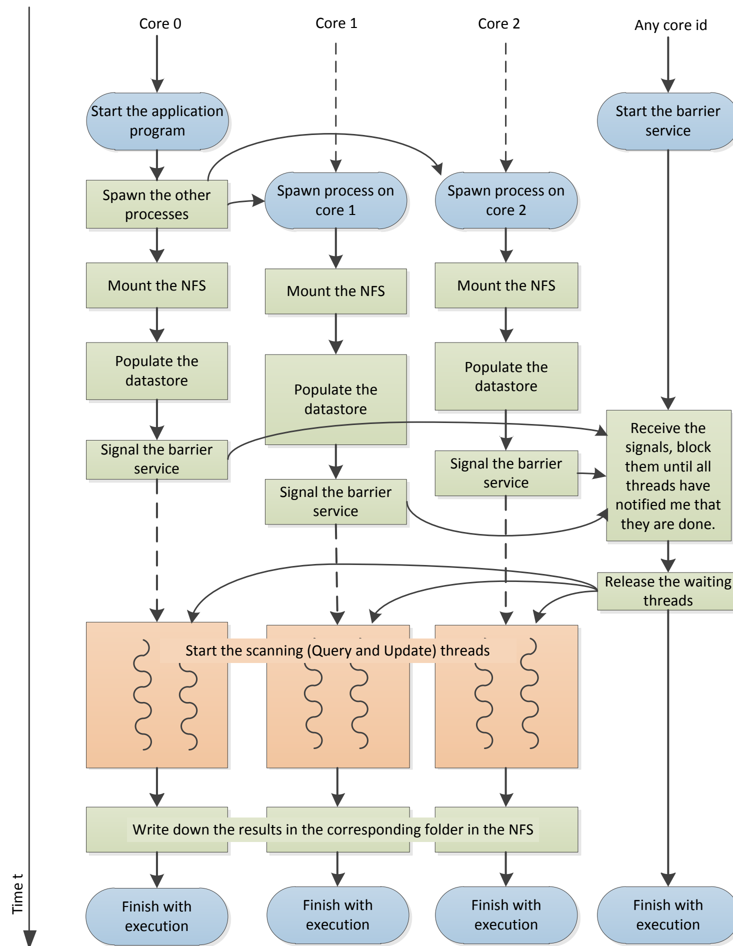


FIGURE 3.3: Design alternative 3: Multiple processes loading and processing the datastore individually synchronizing with each other using the barrier service

execution of the scanning threads, they first try to get in sync with each other by notifying the barrier service, and wait for its signal in order to continue with the execution. As soon as they get it, they start the query and update scanning threads and then write out the collected results and statistics in the appropriate folder in the NFS.

Summary

In this section we presented the issues we encountered when porting the CSCS engine on top of the Barrelfish OS. We analyzed the changes performed on both systems like modifying the memory management, extending the support for the network stack driver and adapting the CSCS engine code to work with Barrelfish. We also presented the design alternatives considered for implementing the scale-up version of the CSCS engine, indicating both the requirements for the implementation and the existing limitations.

4

Experiments and Results

4.1 The workload

For all experiments reported in the thesis we have used the Amadeus workload. Amadeus IT Group[3] is a world-leading service provider for managing travel related bookings. Its core service is the Global Distributed System (GDS), an electronic marketplace that forms the backbone of the travel industry.

The core database in the Amadeus GDS contains dozens of millions of flight bookings. The largest existing view consists of records of each individual booking of a passenger on a particular flight. A record is approximately 350 bytes in size, and consists of 48 attributes. As one travel booking may be related to multiple persons and flights, this view contains hundreds of millions of such records.

Since the data of Amadeus is confidential we had no access to the actual data, but we were provided with the statistics of a dataset of 8 million records. With the help of these statistics we were able to generate various data-store sizes.

The real Amadeus workload was taken from the traces of the database system during one hour period. The queries were highly selective. That is, most queries searched for a specific passenger on a specific flight or for a small set of passengers. All updates involved a single booking only.

Having the workload stats we were also able to play with the number of queries and updates that belong to one batch, keeping the ratio constant. We will refer to these sets of queries/updates as the real Amadeus workload. We also experimented with a few modified workloads. In some experiments we were varying the number of updates processed by the system in the range from 0 to 2048, keeping the number of queries in the system fixed to 2048. Also in this case, all queries and updates were derived from the Amadeus statistics, where there are about 250 updates per

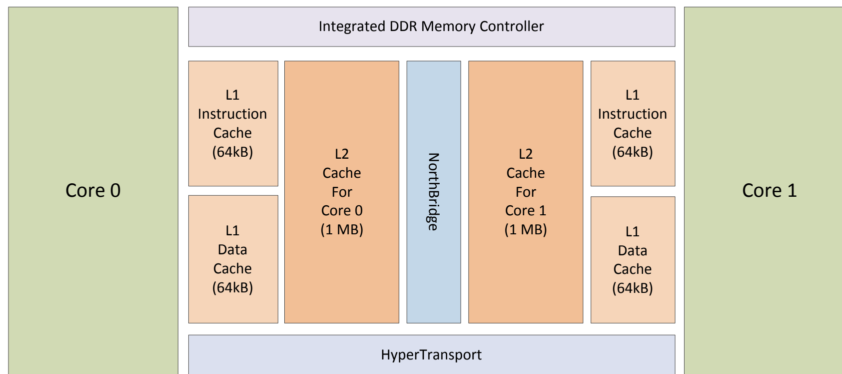


Figure 4.1: AMD Santa Rosa architecture

2000 queries. In other experiments we were just interested in the read only workload, so we only varied the number of queries processed in a batch.

For more information regarding the Amadeus workload and the statistics provided please refer to [27].

4.2 Architecture and specifications of the machines used

We ran our experiments on two AMD machines: AMD SantaRosa 4.1 and AMD Shanghai 4.2. In this section we present a short overview of their specifications along with figures describing their processor architecture.

AMD Santa Rosa - Specifications

The 2x2-core AMD system (Santa Rosa) has a Tyan Thunder n6650W board with 2 dual-core 2.8GHz AMD Opteron 2220 processors, each with a local memory controller. They are connected by 2 HyperTransport links. Each core has its own 1MB L2 cache.

AMD Shanghai - Specifications

The 4x4-core AMD system (Shanghai) has a Supermicro H8QM3-2 board with 4 quad-cores 2.5GHz AMD Opteron 8380 processors connected in a square topology by four HyperTransport links. Each core has a private 512kB L2 cache, and each processor has a 6MB L3 cache shared by all 4 cores.

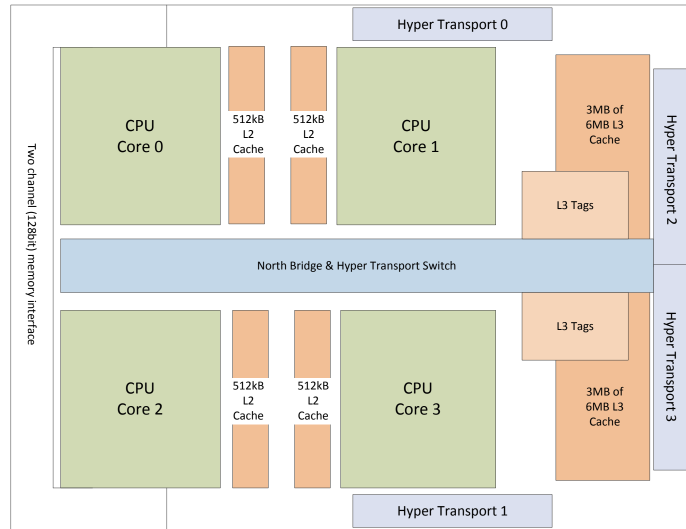


FIGURE 4.2: AMD Shanghai architecture

4.3 Methods and tools for performance analysis

Nowadays, all major processors provide a set of performance counters which capture micro-architectural level information, such as the number of elapsed cycles, cache misses, or instructions executed. Counters can be found in processor cores, processor die, chipsets, or in I/O cards. They can provide a wealth of information as to how the hardware is utilized by the software.

There are two main approaches to event counting: **caliper mode** and performance counter **sampling** [28].

The **caliper** approach reads the event count before and after a performance-critical region of code. This approach measures the number of events, but does not indicate how the events are distributed across the code region. This style of measurement often requires a change to the program source code to take measurements at key points. This is the approach that we used in Barrelfish, using the available API for configuring and reading directly from the performance counter registers. For more details see Appendix B.

In performance **counter sampling**, the event counter is preloaded with a sampling period. The performance measurement hardware counts events until reaching the threshold sampling period and then causes an interrupt recording the part of the code that triggered the event. A profiling tool, then, processes the samples and builds a statistical histogram of how events of a particular type are distributed across the source-lines and/or instructions in the application program. This is the approach that we used in Linux, using the OProfile tool [29].

The complete list of events and the corresponding unit masks that were used in the experiments can be found in Appendix C. Appendix C also contains the list of formulas used in the computation of the derived measurements that will be presented in the following chapters.

4.4 Baseline performance

The following section presents the first set of experiments, measuring the overall performance of the system - throughput and response time. The experiments were conducted on both machines: AMD SantaRosa 4.1 and AMD Shanghai 4.2. Furthermore we present the results from the CSCS engine running on top of both Linux and Barrelfish.

Real workload

We start the analysis by comparing the behavior of the systems when running the real Amadeus workload, varying the number of queries and updates grouped in one batch. We present the exact experiment setup and the results obtained in the following subsections.

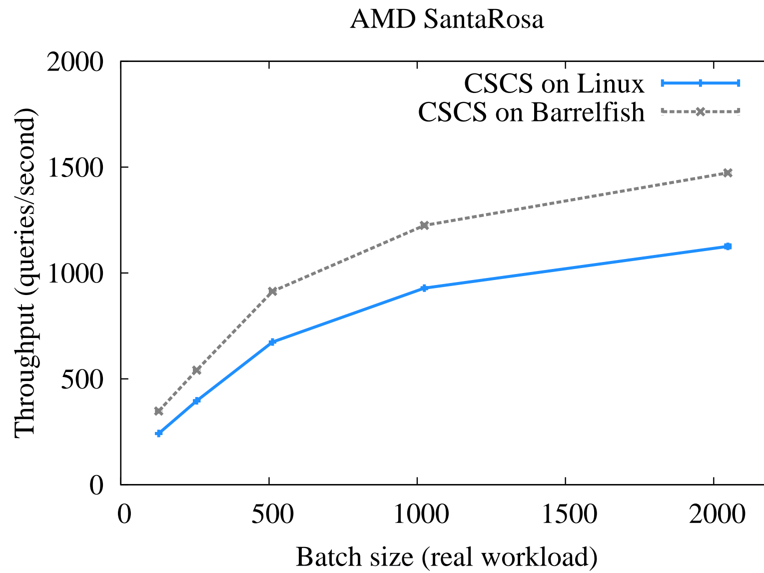
Experiment setup

The system configuration for the experiments can be summarized as following:

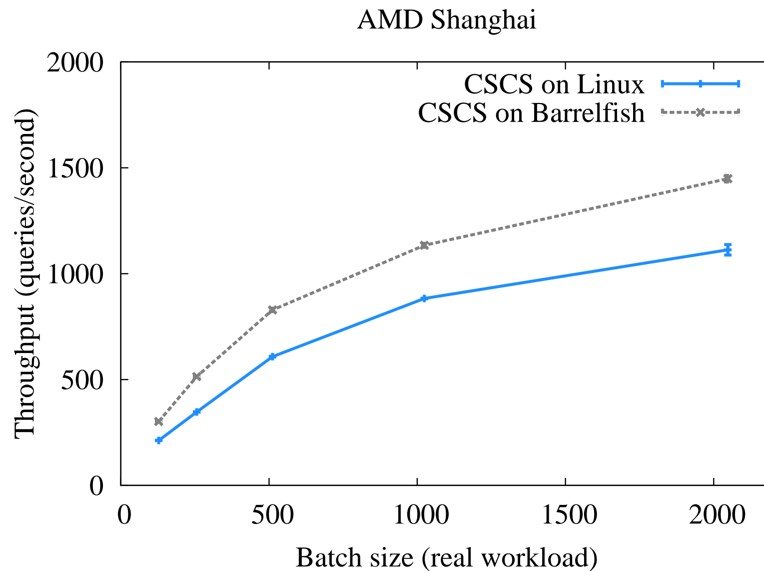
- **database size** = 874 MB (3 million tuples)
- **machines** = {AMD SantaRosa, AMD Shanghai}
- **number of cores** = 1
- **workload**: real workload, varying the batch size
- **(Queries,Updates)** = {(128,16), (256,32), (512,64), (1024,128), (2048,256)}

Experiment results and plots

The results(4.3, 4.4) show that the two systems, CSCS engine running on both Linux and Barrelfish, exhibit the same behavior with the increase of the batch size. In both cases the throughput steadily increases and after a certain threshold, when grouping more than 1024 queries, the throughput continues increasing gradually but with a smaller rate. As expected the corresponding response time increases linearly with the batch size. The same observation is valid for the results of the run on both machines.



(a) AMD SantaRosa Throughput



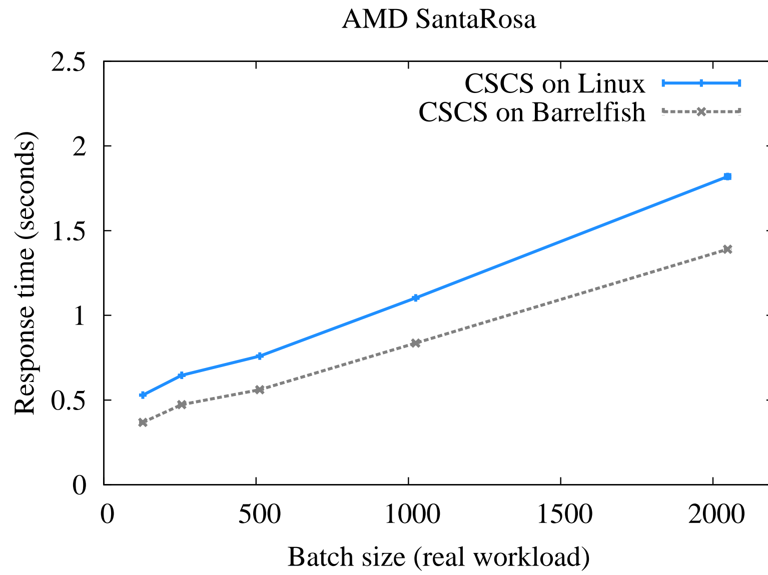
(b) AMD Shanghai Throughput

FIGURE 4.3: Real workload - Throughput

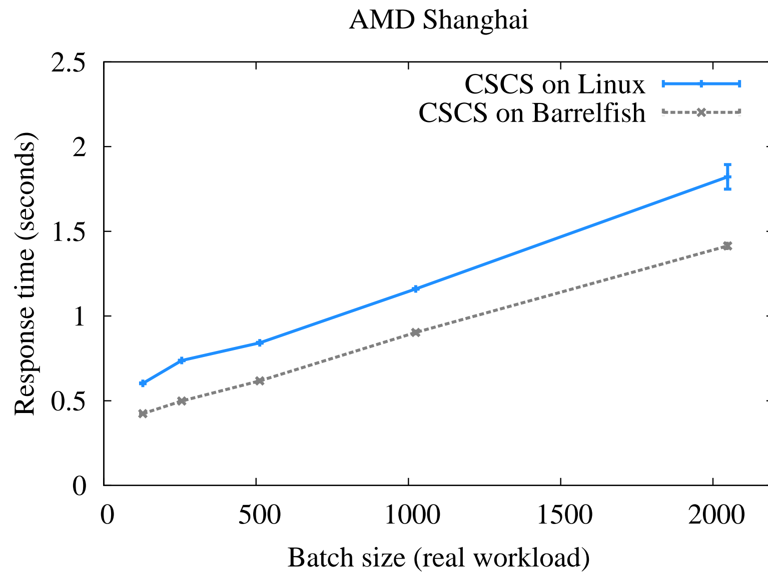
The only noticeable difference is the better performance of the system running on top of Barrelfish. In order to explain this we first need to instrument the main factors known to influence performance [28]. Please refer to section 4.5 for more details regarding this analysis.

Varying the number of updates

We continued the analysis of the overall behavior of the systems by tweaking the workload. The synthetic workload used was created by fixing the number of queries sent to the system, and



(a) AMD SantaRosa Response time



(b) AMD Shanghai Response time

FIGURE 4.4: Real workload - Response time

varying the number of updates processed. In the following subsections we present the exact experiment setup and the obtained results.

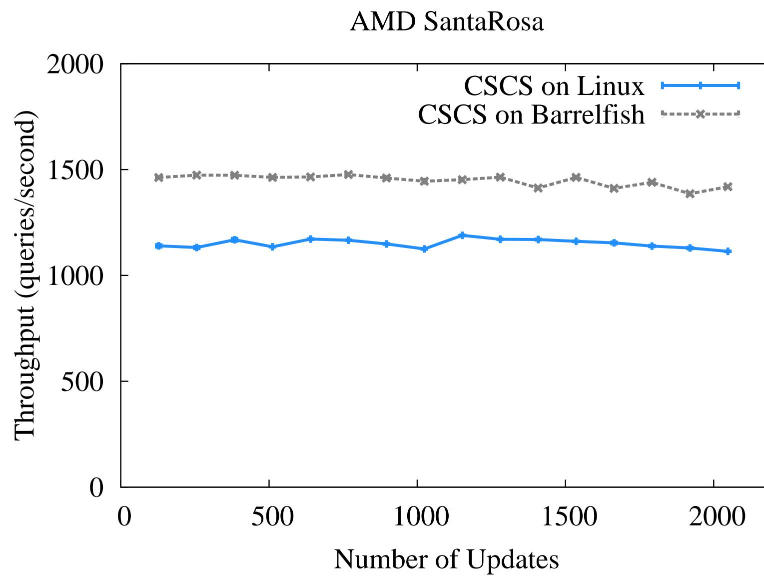
Experiment setup

The system configuration for the experiments can be summarized as following:

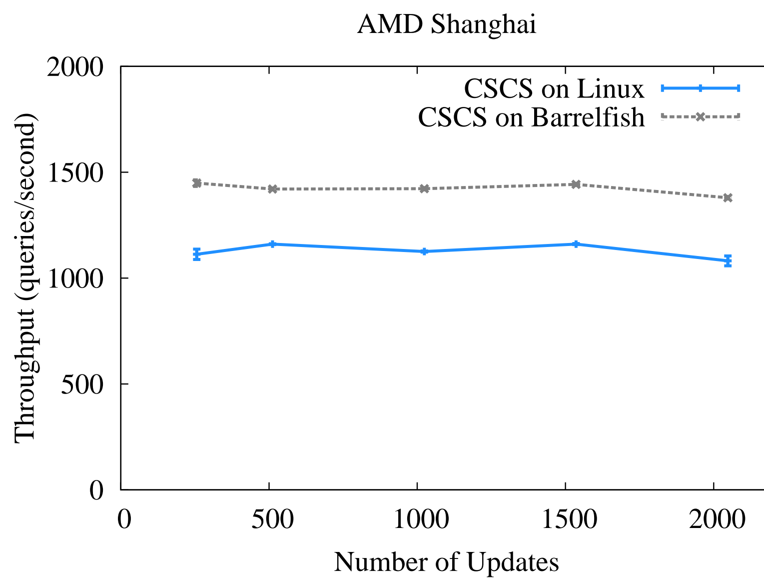
- **database size** = 874 MB (3 million tuples)

- **machines** = AMD SantaRosa, AMD Shanghai
- **number of cores** = 1
- **workload**: synthetic workload, varying the number of updates
- **(Queries,Updates)**={(2048,128), (2048,256), (2048,512), (2048,1024) , (2048,1152), (2048,1280), (2048,1408), (2048,1536), (2048,1664), (2048,1792), (2048,1920), (2048,2048)}

Experiment results and plots



(a) AMD SantaRosa Throughput - Varying updates



(b) AMD Shanghai Throughput - Varying updates

Figure 4.5: Varying the number of updates

From the result plots (Figure 4.5) it can be seen that apart from the previously observed difference in the performance by running the system on top of Barrelfish vs. Linux, the two systems still have the same behavior also in this synthetic workload. In other words the number of updates processed by the system does not influence the overall performance, leaving the throughput in both cases unchanged.

Varying the size of the database

We also wanted to see how both systems react when assigned to operate on various data-store sizes. We fixed the workload to be read only, by processing a batch of 2048 queries at a time. The exact experiment setup and the results obtained are presented below.

Experiment setup

The system configuration for the experiments can be summarized as following:

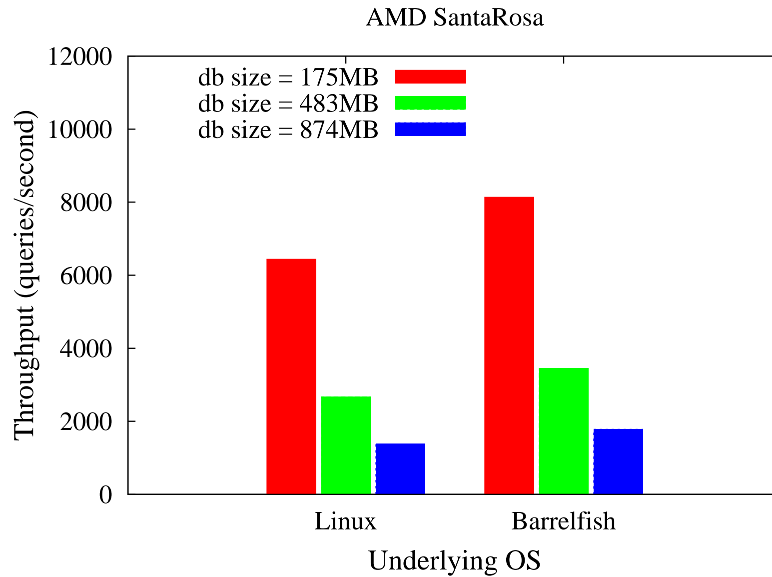
- **database size** = 175MB, 483MB, 874MB
- **machines** = AMD SantaRosa, AMD Shanghai
- **number of cores** = 1
- **workload**: synthetic workload, select queries only
- **(Queries,Updates)** = {(2048,0)}

Experiment results and plots

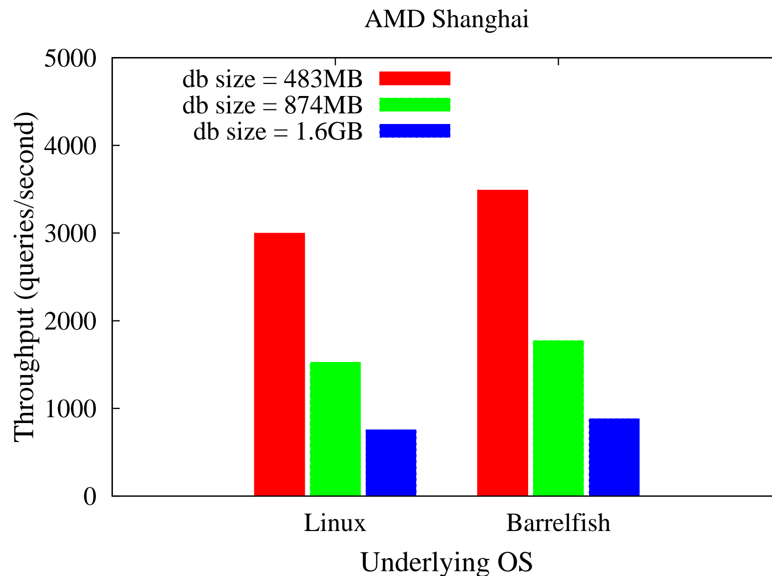
As we can see from the plots in figure 4.6, the performance of both systems is directly proportional to the size of the data-store loaded into the system. As expected, the smaller the size of the scanned data, the smaller the response time and consequently the higher the overall throughput. The same behavior is seen when running the experiment on both AMD machines.

Conclusion

From all experiments presented in this section it can be concluded that the systems (CSCS on Linux and CSCS on Barrelfish) exhibit the same behavior when working on different workloads (real and synthetic), machines, batch and data-store sizes. It was also observed that in all experiments, the CSCS engine running on Barrelfish system was outperforming the one running on Linux.



(a) AMD SantaRosa Throughput



(b) AMD Shanghai Throughput

FIGURE 4.6: Varying the size of the database - Throughput

4.5 Linux vs. Barrelfish performance analysis

This section presents the results of a detailed performance analysis on the execution of the CSCS engine on top of Linux vs. Barrelfish. In particular we were instrumenting the execution of the CSCS engine scanning threads i.e. the overhead of reading up the batch of incoming queries and returning back the results of the requests to the clients as well as loading up the contents of the datastore were ignored.

Experiment setup

All experiments discussed in this chapter were executed with the same experiment setup. The system configuration for the experiments can be summarized as following:

- **database size** = 874MB
- **machine** = AMD SantaRosa
- **number of cores** = 1
- **workload**: real workload
- **(Queries,Updates)** = {(2048,256)}

Results obtained

The calculated results from all measured events [30] are presented in the following table 4.1

TABLE 4.1: Instrumentation results, Linux and Barrelfish, 1 core

Category	Subsystem	Measurements	Barrelfish	Linux
Efficiency	CPU	Instructions per cycle (IPC)	0.94	0.89
		Cycles per instruction (CPI)	1.07	1.13
	Memory	Read data bandwidth (MB/s)	181.99	136.13
		Write data bandwidth (MB/s)	72.43	54.83
		DRAM data bandwidth (MB/s)	255.63	181.50
Memory access	L1 cache	L1 data cache request rate (%)	46.64	41.82
		L1 data cache miss rate (%)	2.00	1.51
		L1 data cache miss ratio (%)	4.29	3.60
		L1 inst cache request rate (%)	39.03	38.30
		L1 inst cache miss rate (%)	0.00	0.03
		L1 inst cache miss ratio (%)	0.00	0.09
	L2 cache	L2 cache request rate (%)	4.39	3.28
		L2 cache miss rate (%)	0.07	0.06
		L2 cache miss ratio (%)	1.61	1.98
Address Translation	DTLB	L1 DTLB request rate (%)	46.60	41.82
		L1 DTLB miss rate (%)	1.81	1.19
		L1 DTLB miss ratio (%)	3.89	2.85
		L2 DTLB request rate (%)	1.81	1.19
		L2 DTLB miss rate (%)	0.17	0.05
		L2 DTLB miss ratio (%)	9.55	3.82

Table 4.1 gives an overview of the performance measurements and the obtained results. The measurements are classified in three different categories: **efficiency**, **memory access** and **address translation**.

The values for IPC and CPI in both cases indicate that the program has efficiently used the available CPU cycles.

In order to evaluate the memory bandwidth utilization, we measured the attainable memory bandwidth on our machine, using the STREAM benchmark [31] and the results obtained are presented in table 4.2:

Table 4.2: STREAM benchmark results

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	3760.34	0.0085	0.0085	0.0085
Scale	3662.15	0.0088	0.0087	0.0089
Add	2524.85	0.0190	0.0190	0.0190
Triad	3676.19	0.0131	0.0131	0.0131

The results from both tables (4.1 and 4.2) indicate that the scanning threads of the CSCS engine have small memory bandwidth utilization, using only around 10% of the available bandwidth.

Good cache behavior is crucial for a good system performance, therefore we also measured the cache for our system. The results in the table for L1 data and instruction cache miss rate and ratio as well as for the L2 cache are very low. This indicates that the program exhibits good spatial and temporal locality when accessing data.

Translation lookaside buffers (TLB) assist the processor to translate virtual addresses to physical addresses by holding the most recently used page mapping information, and with it it helps accelerating the address translation. Therefore it was important to also measure the Data TLB during the execution of the program. The low DTLB miss rate and miss ratio once again confirm that our program exhibits a very good spatial and temporal locality.

Overall, from the results presented we can conclude that, similar to Crescando's statement, the thread running the clock scan on Barrelfish and Linux is CPU bound.

Next we continue with the interpretation of the results in table 4.1 in order to understand the difference in performance of the CSCS running on top of Linux vs. Barrelfish. In all measurements the calculated values are almost identical with small variations due to noise. The only exception are the memory bandwidth utilization results.

The memory bandwidth utilization is calculated using the following formula[1].

$$\text{Read bytes transferred} = (\texttt{system_read} * \text{period}) * 64 \tag{4.1}$$

Furthermore we use the `cpu_clocks_unhalted` event to measure the elapsed time. The formula that gives us the elapsed time is the following:

$$\text{Seconds} = \frac{(\texttt{cpu_clocks_unhalted} * \text{period})}{\text{clock frequency}} \tag{4.2}$$

Bandwidth is then calculated as the number of bytes that were transferred in a unit of time, i.e. using the following formula:

$$\text{Read bandwidth}\,(B/s) = \frac{\text{Read bytes transferred}}{\text{Seconds}} \tag{4.3}$$

The intermediary results, such as the bytes transferred and the time elapsed, were not presented in the main result table. Since they are the main factors influencing the final value of the bandwidth utilization, we present them in the following table:

TABLE 4.3: Intermediary bandwidth utilization results

Measurement	Barrelfish	Linux
Read bytes transferred (MB)	254.79	274.97
Write bytes transferred (MB)	101.40	110.75
DRAM bytes transferred (MB)	357.87	366.63
Seconds	1.43	1.81

As we can see from the table 4.3, the amount of data transferred in both systems is almost identical with small variations due to noise and lower precision granularity in the Linux results because of the sampling method when collecting event values. This leaves the CPU usage, as the main factor influencing the difference in the performance of the systems.

Our previous performance analysis indicated that our system processing the queries is CPU bound. Therefore, CPU being the scarce resource, it is a sound guess that the differences in performance are a result of different utilization of the CPU when the CSCS is executed on Linux and Barrelfish. We investigated this in more details with another round of experiments, measuring the CPU usage by each executable getting hold on the CPU.

[1]For brevity only the formulas used for calculating read bandwidth utilization are presented here, all the other formulas can be found in Appendix C.

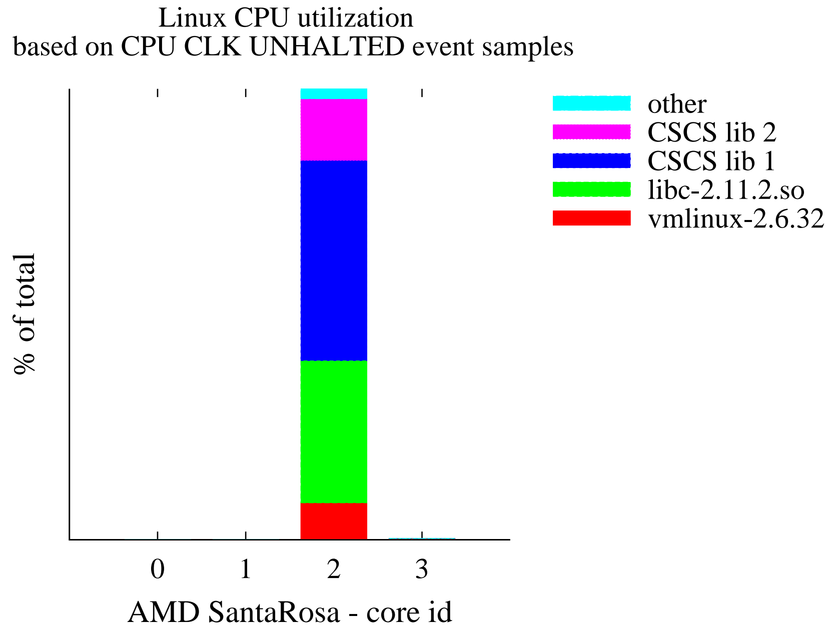


FIGURE 4.7: CSCS on Linux: CPU usage breakdown

Figure 4.7 shows the cpu utilization breakdown as measured by OProfile on Linux. The effects of libc++ library are dispersed over either the stdlibc or the application code itself, as it consists mainly of header files.

Further analysis of the lib C usage showed that the string compare function `strncmp` was the place where the CPU spent most of its time. We measured specific segments of the code implementing the ClockScan algorithm 2.1.3. The execution time breakdown results confirm that the run on Linux spends the difference in execution time in evaluating the non-indexed predicates, which mainly involves string comparison (Figure 4.8).

The reason for the difference in performance for string comparison lies in the usage of different standard C++ libraries in Linux and Barrelfish, as explained in Section 3.1.

4.6 Performance on multiple cores

The following section presents the set of experiments measuring the overall performance of the system when using multiple cores available on the machine. The experiments were conducted on AMD Shanghai 4.2 and we present the results from ClockScan on Column Stores running on top of both Linux and Barrelfish.

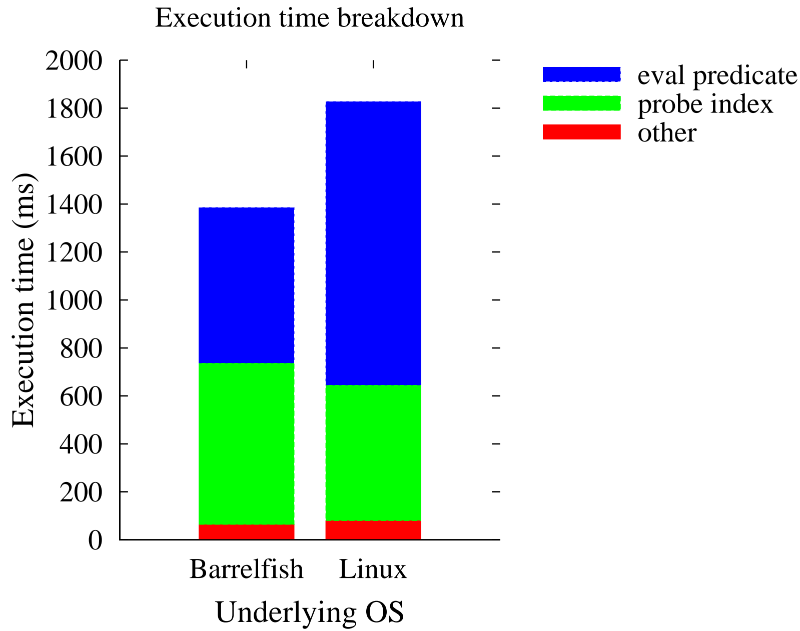


FIGURE 4.8: Execution time breakdown

Experiment setup

All experiments discussed in this chapter were executed with the following experiment setup:

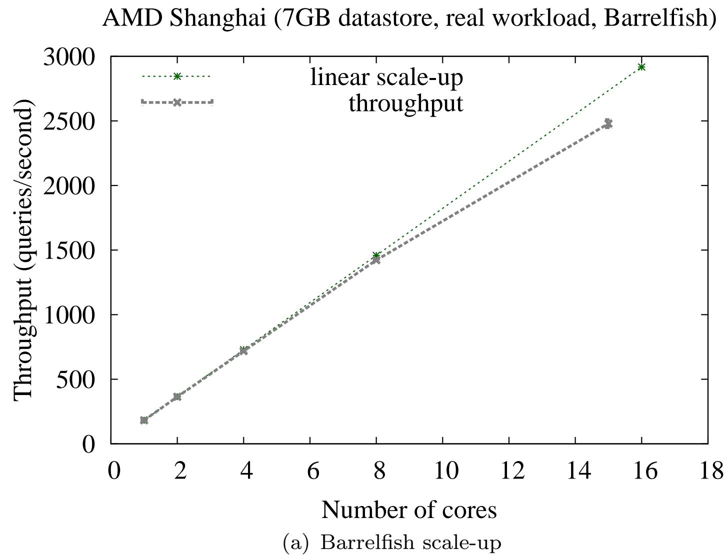
- **total database size** = 6.9GB
- **size of database partitions** = {6.9GB, 3.5GB, 1.8GB, 874MB, 438MB}
- **number of cores** = {1, 2, 4, 8, 16}
- **machine** = {AMD Shanghai}
- **workload**: real workload
- **(Queries,Updates)** = {(2048,256)}

Note: For the scale up experiments we fixed the size of the data-store on which the system operates on 7GB. When running on multiple cores, each core was set to operate on a separate partition of the dataset. The dataset is equally partitioned among all the cores used in the experiment run.

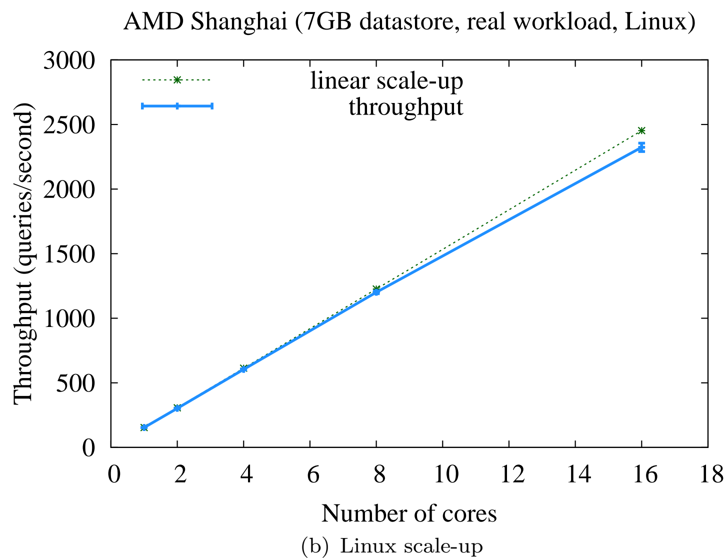
Experiment results and plots

The results obtained from the experiments are presented in figures 4.9 and 4.10. Figure 4.9 presents the performance of the systems with respect to the calculated linear scalability line.

The linear scalability line was obtained by interpolating the values based on the performance measured when running on one core. These figures indicate that in both cases, the performance



(a) Barrelfish scale-up



(b) Linux scale-up

FIGURE 4.9: Scale-up performance on AMD Shanghai

has the same behavior scaling almost perfectly linear with the increasing number of cores.

Figure 4.10 presents the difference in performance of the ClockScan on Column Stores on top of Barrelfish and Linux.

As discussed earlier, also in figure 4.10 we can see the difference between the performance achieved when running the CSCS engine on top of Linux and Barrelfish. We can also see that it actually increases with the number of cores. The previous two figures, however, showed that both of them scale almost linearly with the number of cores. So we can conclude that the

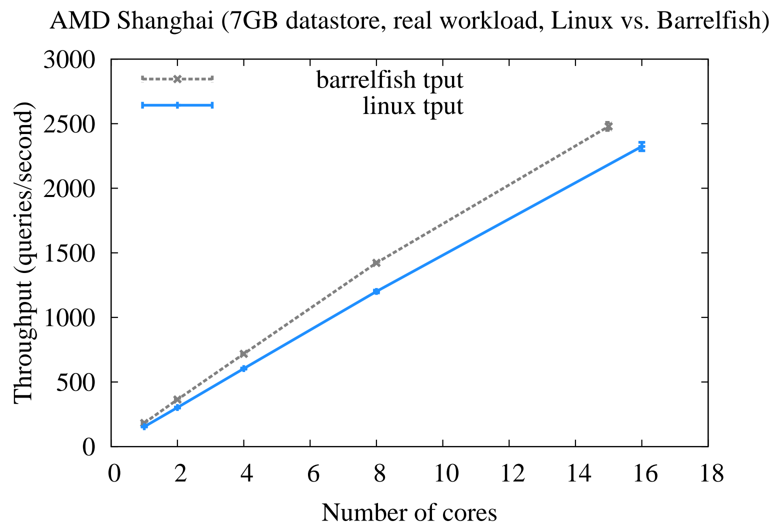


FIGURE 4.10: Scale-up performance comparison Linux vs. Barrelfish on AMD Shanghai

increase of the difference in throughput results from the accumulation of the initial differing in the performance.

Conclusion

From all experiments presented in this section it can be concluded, that CSCS has the same behavior on Linux and Barrelfish even when executed on multiple cores. The scalability achieved both on top of Linux and Barrelfish was close to the ideal scalability with the number of cores.

4.7 Scale-up analysis

This section presents the results of a detailed performance analysis on the execution of the CSCS engine on top of Barrelfish on the AMD Shanghai machine. We are comparing the values of the most basic performance measurements, as specified in [28], when the program utilizes 1 and 16 cores.

Experiment setup

All experiments discussed in this chapter were executed with the same experiment setup. The system configuration for the experiments can be summarized as following:

- **database size** = 7GB
- **machine** = AMD Shanghai

- **number of cores** = {1,16}
- **workload**: real workload
- **(Queries,Updates)** = {(2048,256)}

Results obtained

The calculated results from all measured events [32] are presented in the following table 4.4

Table 4.4: Instrumentation results, Barrelfish, multiple cores

Category	Subsystem	Measurements	1 core	16 cores
Efficiency	CPU	Instructions per cycle (IPC)	0.935	1.014
		Cycles per instruction (CPI)	1.069	0.986
	Memory	Read data bandwidth (MB/s)	181.99	179.42
		Write data bandwidth (MB/s)	72.43	177.89
		DRAM data bandwidth (MB/s)	255.63	307.51
Memory access	L1 cache	L1 data cache request rate (%)	46.64	47.08
		L1 data cache miss rate (%)	2.00	2.14
		L1 data cache miss ratio (%)	4.29	4.55
		L1 inst cache request rate (%)	39.03	29.34
		L1 inst cache miss rate (%)	0.00	2.02
		L1 inst cache miss ratio (%)	0.00	6.89
	L2 cache	L2 cache request rate (%)	4.39	4.79
		L2 cache miss rate (%)	0.07	0.10
		L2 cache miss ratio (%)	1.61	2.04
	L3 cache	L3 cache request rate (%)	5.48	7.49
		L3 cache miss rate (%)	0.03	0.17
		L3 cache miss ratio (%)	0.50	2.23
Address Translation	DTLB	L1 DTLB request rate (%)	46.60	47.08
		L1 DTLB miss rate (%)	1.81	1.20
		L1 DTLB miss ratio (%)	3.89	2.55
		L2 DTLB request rate (%)	1.81	1.20
		L2 DTLB miss rate (%)	0.17	0.22
		L2 DTLB miss ratio (%)	9.55	18.18

Analysis and interpretation of the results

Table 4.4 gives an overview of calculated results of the basic performance measurements. The measurements, as previously, are classified in three different categories: **efficiency**, **memory access** and **address translation**.

Also in this experiment, the values for IPC and CPI in both cases indicate that a good amount of work was done in the given time interval.

The results for L1 data and instruction cache miss rate and ratio as well as for L2 and L3 caches are very low. These, together with the low DTLB miss rates confirm that the program exhibits good spatial and temporal locality when accessing data, even when running on multiple cores.

Eventually we can conclude that the program remains CPU bound also in the scale up experiments.

Comparing the execution of the program running on 1 and 16 cores, one can see that the corresponding values in the table are almost identical, differing only slightly due to noise. The only notable difference is in the results of L3 cache miss rate and ratio. As expected, the miss rate has increased four-fold when executing the same application on all 16 cores, because now 4 cores are accessing the same L3 cache simultaneously with the same miss rate.

4.8 NUMA support

Since the architecture of the machine has a NUMA nature, we were interested in the impact on the overall performance when the CSCS engine was made NUMA-aware in contrast to the runs when it was not. In the following section, we will start with a small overview of the NUMA support available on the two underlying systems Linux and Barrelfish; and then conclude with a short analysis and discussion.

Shortly, **NUMA** stands for Non-Uniform-Memory-Allocation. In the simplest form of NUMA, a processor has a local memory that is much cheaper to access than memory local to other processors. A **NUMA node** is a set of CPUs that have equal fast access to some memory chunk using a memory controller. The memory of different NUMA nodes on a machine is connected via fast interconnects. In a cache coherent machine the remote memory can be used the same way as a local, the only difference being the slower access due to the interconnect.

Barrelfish and Linux provide different support for getting the information about the NUMA nature of the machine and to make applications run in a NUMA-aware mode. Linux offers the NUMA API [33]: `numactl` and `libnuma`. They both provide mechanisms for specifying NUMA policies for either memory regions or processes (threads). `numactl` is a command line tool to run processes with a specific NUMA policy, and it is the one we used in Linux to bind the specific application and its corresponding threads to run at a specific core and thus use the memory belonging to its local NUMA node. Barrelfish on the other hand, provides a function call `set_skb_affinity()`, part of the Barrelfish System Knowledge Base (SKB). It can be invoked

from within the application itself to set the NUMA affinity of the current process to use the local NUMA node for subsequent memory allocations.

AMD Shanghai NUMA architecture

The experiments in the following section were performed on the AMD Shanghai machine. The NUMA architecture of it is best presented with Figure 4.11. As we can see there are four NUMA nodes, and they are all interconnected with each other via HyperTransport^{TM} links.

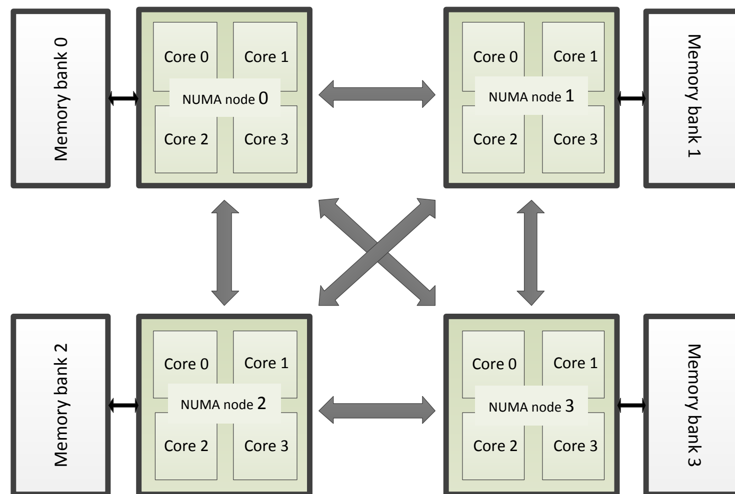


FIGURE 4.11: AMD Shanghai NUMA architecture

4.9 NUMA effect analysis

This section presents the results of the experiments performed on the CSCS engine running on top of Barrelfish measuring the effect of NUMA awareness on the behavior of the system and its overall performance.

Experiment setup

All experiments discussed in this chapter were executed with the same setup. The system configuration can be summarized as following:

- **database size** = 438MB
- **machine** = AMD Shanghai

- **number of cores** = 16
- **workload**: real workload
- **(Queries,Updates)** = {(2048,256)}

Experiment theory: Events gathered and formulas used

The results presented below are based on the measurements of the following events:

TABLE 4.5: Events used in the NUMA effect analysis

Event number	Event abbreviation	Event description
0x1E0	`CPU to DRAM Requests to Target Node`	counts DRAM Reads/Writes generated by the cores on the local node to the targeted node
0x0E9	`CPU/IO Requests to Memory/IO`	reflect request flow between units and nodes, as selected by the unit mask

The results presented in the following section are basically the percentage of registered events with a specific unit mask out of the total measured events.

In other words the formulas used are the following:

$$\begin{aligned} \text{local_node_to_all_nodes} &= \texttt{local_node_to_node0} + \texttt{local_node_to_node1} \\ &+ \texttt{local_node_to_node2} + \texttt{local_node_to_node3} \end{aligned} \tag{4.4}$$

$$\text{Local to Node 0} = \frac{\texttt{local_node_to_node0}}{\text{local_node_to_all_nodes}} \tag{4.5}$$

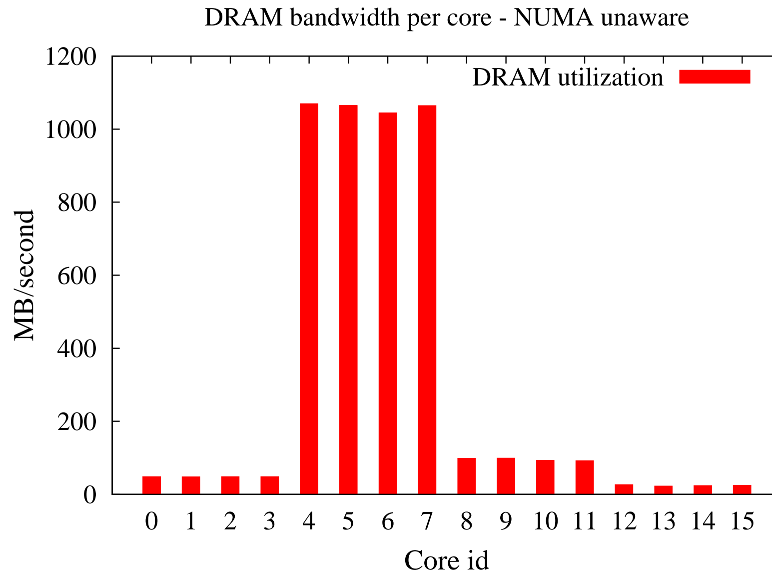
Similarly for each NUMA node i:

$$\text{CPU to Memory}_i = \texttt{cpu_mem-local_local}_i + \texttt{cpu_mem-local_remote}_i \tag{4.6}$$

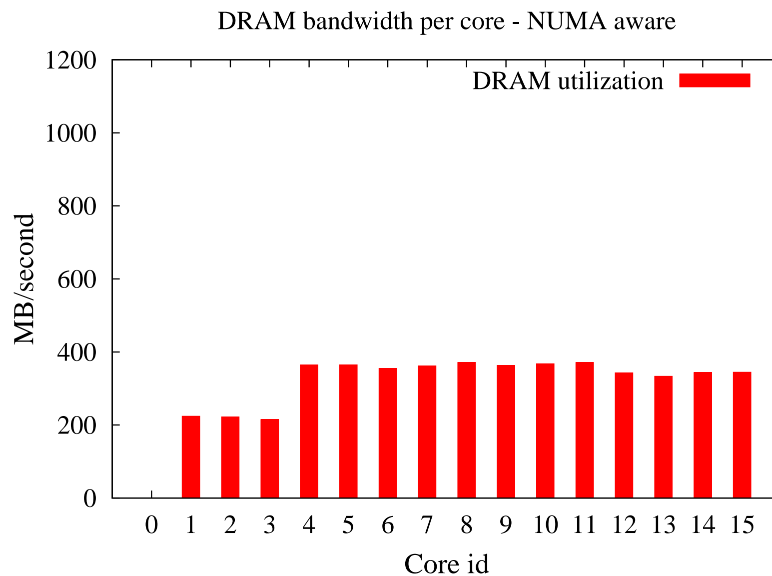
$$\text{CPU to Memory, Local to local}_i = \frac{\texttt{cpu_mem-local_local}_i}{\text{CPU to Memory}_i} \tag{4.7}$$

Results obtained

Figure 4.12 shows that the memory controllers in each switch are not equally utilized. The biggest load is on the second NUMA node (cores 4-7). As explained earlier, this is because the database is not aware of the NUMA architecture present in the machine. The program just invokes regular `malloc` to allocate memory, hence the memory is allocated in a sequential fashion.



(a) NUMA ignorant DRAM utilization

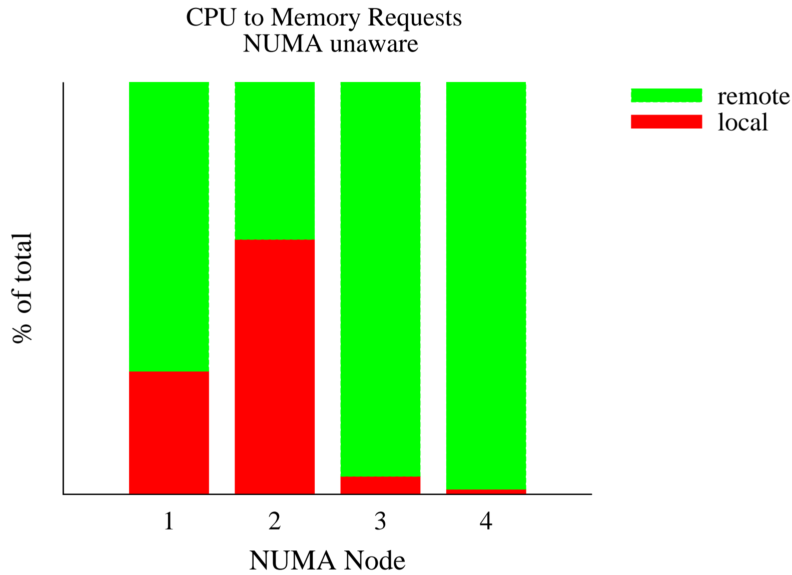


(b) NUMA aware DRAM utilization

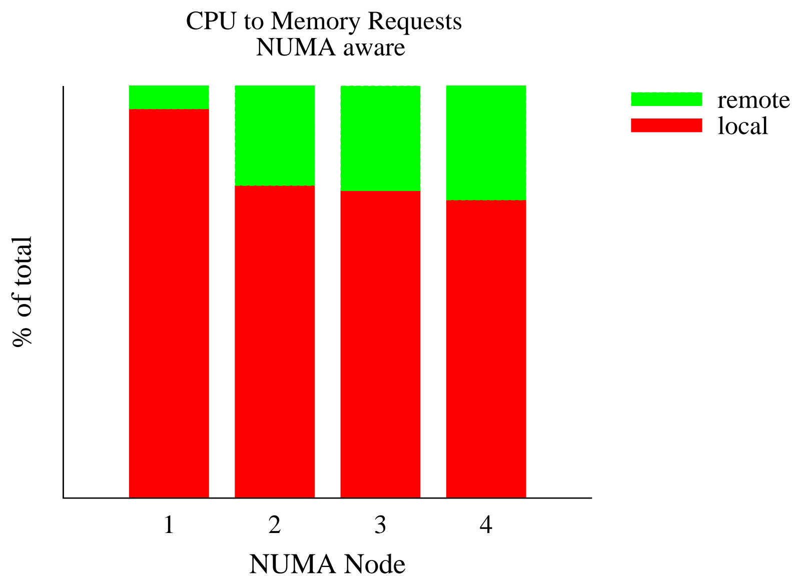
FIGURE 4.12: Effect of NUMA awareness on DRAM utilization

Further analysis showed that the memory allocated in the first node is mainly for the dispatchers' code and some other OS-related NUMA unaware data-structures and programs. Consequently, the data-store data-structures were mainly allocated in the second and partially on the first and third NUMA node.

Another round of experiments was executed after modifying the CSCS engine code to become NUMA-aware and changing the dispatchers to be NUMA-sensitive if the user demands so. The results of these experiments are presented inline (Figures 4.12, 4.13, and 4.14) with the results of the NUMA-unaware runs for better comparison of the NUMA effects.



(a) NUMA ignorant DRAM utilization

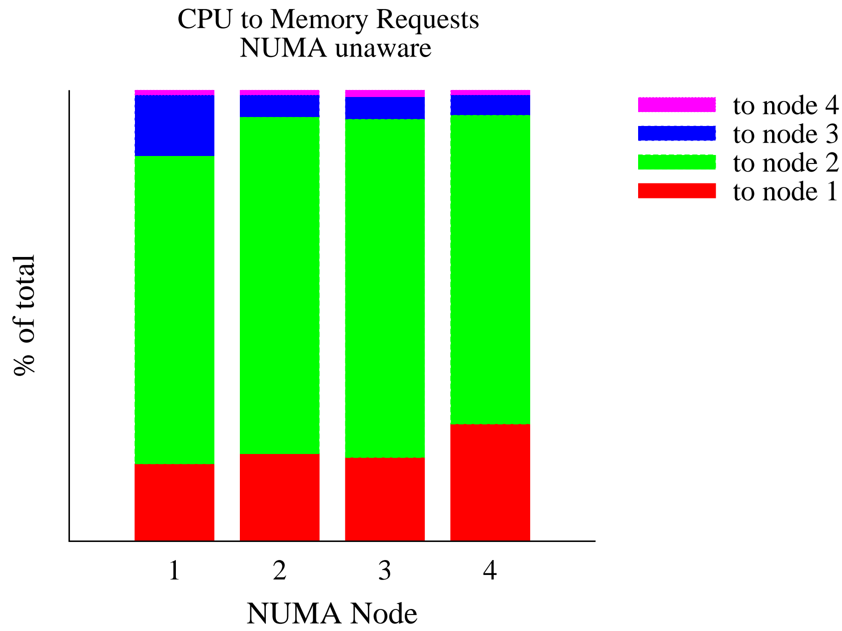


(b) NUMA aware DRAM utilization

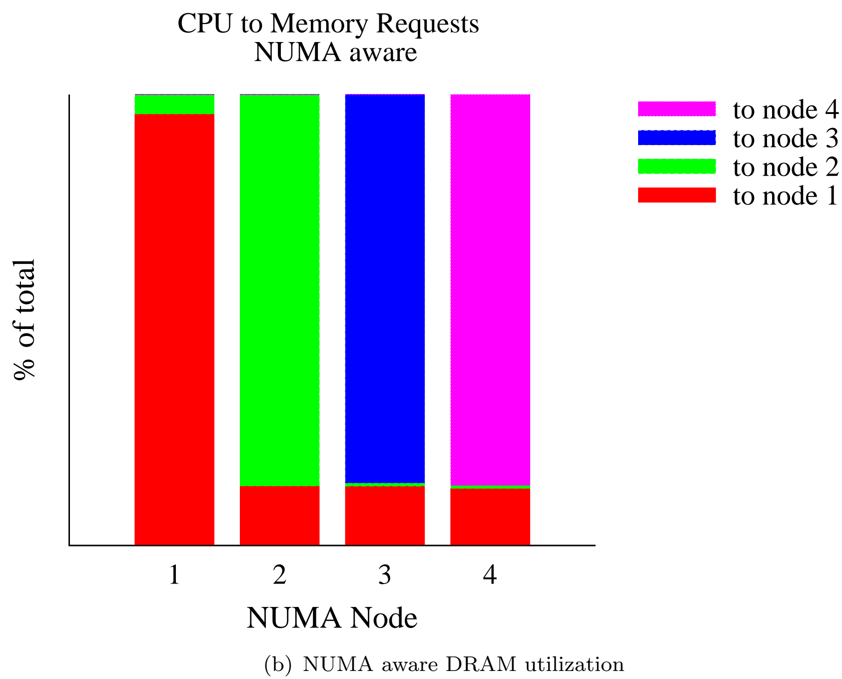
FIGURE 4.13: Effect of NUMA awareness: Local to remote NUMA node access ratio

Figures 4.13 and 4.14 show results per memory controller i.e. per NUMA node. The results presented are the average of the values measured at each core belonging to the same processor.

From figure 4.13 we can observe that a small percentage of the memory access still involves remote memory, even with the NUMA aware database. The results from a more detailed experiment (Figure 4.14) show that most of the remote access is to the first NUMA node. The reason for this is that some of the core Barrelfish functionality is stored and executed on core 0 (and thus NUMA node 1), so it will remain a constant overhead on the interconnect.



(a) NUMA ignorant DRAM utilization



(b) NUMA aware DRAM utilization

FIGURE 4.14: Effect of NUMA awareness: local to other NUMA nodes access ratio

All other measured events on the multi-core execution of the program were independent of the underlying NUMA architecture. In other words, there is no visible difference in the value measured among the cores[2].

Lastly, we measured how much NUMA awareness influences the performance of the system. The experiment setup is the following:

- **database size** = total of 6.4GB, per core 438MB
- **machine** = AMD Shanghai
- **number of cores** = 15
- **workload**: real and read-only workload
- **(Queries,Updates)** = {(2048, 256), (2048, 0)}

The results that we obtained are presented in figure 4.15. As we can see the effects on NUMA are almost negligible for the CSCS engine scanning threads. The reason for that is the low memory bandwidth utilization, as presented in section 4.5, and the fact that all NUMA nodes are interconnected with each other 4.11 so the penalty for accessing memory from any remote node is still low.

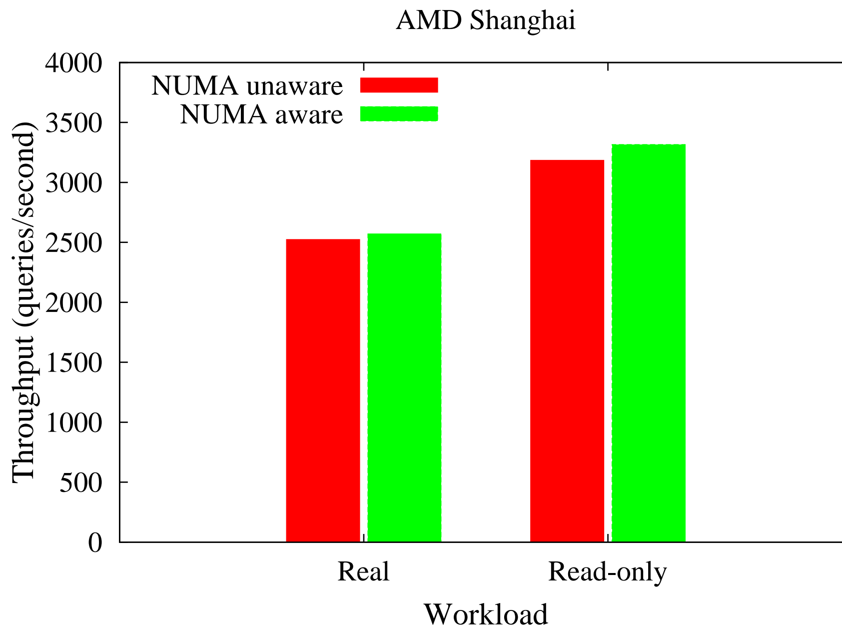


FIGURE 4.15: Effect of NUMA awareness on system performance

One problem that occurs when running the NUMA-aware CSCS engine on top of Barrelfish is that the NUMA policy is so strictly obeyed that once we run out of memory on the first NUMA

[2] Please find all plots on the wiki page [34]

node the program crashes. This is why we do not run the application on core 0. When running on all cores we simply run out of memory on NUMA node 1, as there are some additional programs, that are either NUMA unaware or are simply fixed to execute on core 0. Initially that was also the case with the dispatchers' code. With some minor modifications they are now allocated locally on the NUMA node where they belong, if the user demands so.

Conclusion

Looking at the results and the analysis of the experiments we can infer that the choice of the operating system is irrelevant: apart from the performance gain due to using different C and C++ libraries, there does not seem to be there any specific advantage of using Barrelfish over Linux, but neither the other way round.

Barrelfish displays the same behavior characteristics as Linux and can therefore be safely used as a basis for future development of a more complex system like a fully functional database. System, that will be designed to make use of most of the important services offered by Barrelfish.

5

Conclusion

To conclude we are going to give a brief summary of the thesis contribution and a short overview of the points we plan to investigate in the future.

5.1 Thesis contribution

As stated in the motivation, the main goal of this thesis was to explore the possibility for starting an interesting research direction - *database-operating system co-design* with the Barrelfish OS and the CSCS engine as building blocks. A short introduction of the relevant aspects of the systems was provided in Chapter 2. It was meant to provide an overview of the key points behind the whole motivation as well as to introduce the reader to concepts used in the other sections of the thesis.

We ported the CSCS engine and its dependencies on top of Barrelfish. As explained in Chapter 3, porting it required changing some parts in both systems, such as: modifying the memory management, extending the support for the networking stack driver, adapting the CSCS engine code to work on Barrelfish as well as extending it to work on multiple cores. Then, we did an extensive experiment analysis (Chapter 4), comparing the performance of the new system with the more traditional variant (the CSCS engine executing on top of Linux). We further characterized their behavior by monitoring a range of performance events that helped us profile both systems and compare them in a more fine grained level. We performed a number of experiments that excersized the system's scalability properties. In most of the experiments, the compound system's performance tests were performed on two different architectures: AMD SantaRosa and AMD Shanghai. The results confirmed that the system exhibits the same behavior when running on Linux and Barrelfish, with an increased throughput on Barrelfish that was found to be due to several factors: kernel, GCC compiler version, but mainly due to the different C and C++ library implementations.

With this, we believe we answered the questions of our problem statement (Section 1.2). As a main conclusion, we can state that the database is a well written program that is portable to different OSes and continues being CPU bound, but is still too simple to exercise and utilize the main functionalities for the OS-database co-design offered by Barrelfish: the SKB and the scheduler. Barrelfish, on the other hand, not only resembles the behavior of the more-mature Linux when hosting the application, but also outperforms it with a better use of the available libraries.

5.2 Future work

To give a short insight in the future plans, we classify our ideas in two parts: points that will be in our focus in the short run, and goals that we aim to investigate in the long run.

In the short run:

1. As soon as the interrupts are working, and sufficient support is provided for the performance counter measurements for the Intel machines, we plan to run more experiments on different architectures including the AMD 48 core machine (Magny-cores), Intel machines, or on more diverse machines like the Single Chip-Cloud (SCC) (non-cache coherent machine).
2. We are also hoping to see more interesting results when running the scale-up version of the CSCS-engine on a machine with more than 16 cores. We are interested in discovering the point when we are going to reach another bottleneck like memory bandwidth or bigger NUMA effects that are going to degrade the linear scale-up.
3. There is ongoing work in improving the support for fast loading of the datastore, using bulk transfer instead of RPC calls in the Network driver support.
4. We also plan to make the set_skb_affinity more flexible. As discussed earlier, the current implementation does not allow the program to continue with the execution if there is not enough memory in the “preferred” NUMA node. A possible alternative is to allow memory to be borrowed from another node.
5. Furthermore, we also want to check the performance of the system running on multiple cores, if implemented using threads. This still remains one design alternative that will be investigated once we remove the current limitations.

In the long run:

1. At this point, both Barrelfish and the CSCS engine have the necessary functionality to operate together, but are still in the development phase where we can build up the additional functionalities bearing in mind the communication between them. One aspect is extending the CSCS engine to a real datastore engine (with queries and updates coming in, batching them and distributing them to the processing nodes, and returning back the results to the users). That would also require additional detection and optimization of all possible bottlenecks in the new model. Furthermore, as we upgrade the datastore it will have to start using the smart-scheduling powers of Barrelfish, and the knowledge provided by the SKB service.

2. Another venue for future work will be to improve the support in Barrelfish for performance counter measurements. In particular, it should be able to detect what portion of the events resulted from one particular application or subsystem.

3. Furthermore, we want to extend the SKB to be able to offer more online information about the available resources.

Appendix A

CMake to Hake translation

This appendix gives a short insight of some of the steps necessary when translating a CMake[23] file into Hake[24].

Setting up the values in `Configure.h`

In `CMake`:

```
### SET THE VALUES FOR THE Configure VARIABLES
if( something )
  set (MY_VARIABLE ``true'')
else()
  set (MY_VARIABLE ``false'')
endif

### CREATE THE Configure.h FILE
```

Listing A.1: Configuring the values for the Configure.h.in file

In `Hake`:

Manually set the values for the Configure.in variables directly in the Configure.h files

Copy files in the binary directory

In `CMake`:

```
### COPY FILES AND MAKE SYMBOLIC LINK TO OTHER DIRECTORIES
execute_process(
  COMMAND cp ${PROJECT_SOURCE_DIR}/file_name.bin ${PROJECT_BINARY_DIR}
   ...
```

```
COMMAND ln -f -s ${PROJECT_SOURCE_DIR}/directory_name/ -t ${PROJECT_BINARY_DIR}
    )
```

LISTING A.2: Copying files and creating symbolic link to other directories

In `Hake`:

Manually copy the files from the source to binary directory (`/home/netos/tftpboot/gicevaj`). it is recommended that you put the large directories in the NFS directory `/local/nfs/gicevaj` on the Emmentaler machine.

Building a library

In `CMake`:

```
### BUILD LIBRARY A
add_library (lib_a SHARED
  "${PROJECT_SOURCE_DIR}/a/file_1a.cpp"
  "${PROJECT_SOURCE_DIR}/a/file_2a.cpp")
### BUILD LIBRARY B
add_library (lib_b SHARED
  "${PROJECT_SOURCE_DIR}/b/file_1b.cpp"
  "${PROJECT_SOURCE_DIR}/b/file_2b.cpp")
```

LISTING A.3: Creating a library in `CMake`

In `Hake`:

Split the CMake into several Hake files. In particular when building a library, create a separate folder in `barrelfish/lib/` for example `lib_A`. Include all source files in that directory, and all header files in the corresponding `barrelfish/include/` directory. Each `barrelfish/lib` subdirectory has its own Hake file. The Hake file has a form:

```
[build library {
      target = "lib_a",
      cxxFiles = [ "file_1a.cpp", "file_2a.cpp" ],
      architectures = "x86_64",
      addIncludes = [ "file.h" ]
}]
```

LISTING A.4: Creating a library in `Hake`

Similarly for `lib_b`.

Creating an executable

In `CMake`:

```
### BUILD THE TEST EXECUTABLES
add_executable (test_file.exe "${PROJECT_SOURCE_DIR}/test/test_file.cpp")
### LINK THE TEST EXECUTABLES
target_link_libraries (test_file.exe lib_a lib_b)
```

Listing A.5: Creating an executable in `CMake`

In `Hake`:

Put the `text_file` program file to the designated place, usually `/barrelfish/usr/test_name/` and create a separate Hake file for the executable that we want to create

```
[ build application {
      target = "test_file",
      architectures = "x86_64",
      cxxFiles = [ "test_file.cpp" ],
      addLibraries = [ "lib_a", "lib_b", any additional barrelfish libraries]
}]
```

Listing A.6: Creating an executable in `Hake`

Hake limitations

Still unresolved how to do it in Hake:

compile something → execute it → generate new .cpp files → compile them into a new library → link that one with the rest of the executables.

Appendix B

Changes performed on the CSCS engine

This appendix presents a short overview of some of the changes performed on the CSCS engine when porting it on Barrelfish. Each examples provides a code excerpt from Linux implementation and its corresponding Barrelfish equivalent replacement.

Measuring execution time

Getting the duration of execution time in Linux:

```
#include "Utils.h"
...
timeval startTime, stopTime;
gettimeofday(&startTime, NULL);
...
gettimeofday(&stopTime, NULL);
unsigned int msecs = (stopTime.tv_sec - startTime.tv_sec ) * 1000
```

Listing B.1: Acquiring duration of execution in Linux

Doing the exact same thing in Barrelfish requires:

```
#include <bench/bench.h>
#include "arch/x86/bench/bench_arch.h"
...
bench_arch_init();
cycles_t startTime = bench_tsc();
...
cycles_t stopTime = bench_tsc();
```

```
uint64_t msecs = bench_tsc_to_msc(stopTime − startTime);
```

Listing B.2: Quiring duration of execution in Barrelfish

Handling threads

Creating and joining threads, and working with synchronization locks on Linux:

```
pthread_mutex_t sync_lock;
pthread_mutex_lock(&sync_lock);
pthread_create(&t1, NULL, &runThread1, (void*)qp);
pthread_join(t1, NULL);
```

Listing B.3: Working with threads in Linux

Doing the same thing in Barrelfish requires:

```
#include "barrelfish/threads.h"
...
thread_mutex sync_lock;
thread_mutex_lock(&sync_lock);
t1 = thread_create(runThread1, (void*)qp);
thread_join(t1, NULL);
```

Listing B.4: Working with threads in Barrelfish

Working with NFS

In order to read from NFS, the following must be done in the program:

```
#include <vfs/vfs.h>
#include <vfs/vfs_path.h>
#include <errors/errno.h>
...
errval_t err;
vfs_init();
...
err = vfs_mount("/tmp","nfs://10.110.4.4/local/nfs/nfs_folder/datastore);
if (err_is_fail(err)) {
    printf("vfs mount failed\n");
    return 1;
}
```

Listing B.5: Handling NFS files in Barrelfish

Spawning multiple instances of the same application

In order to spawn multiple instances of the same application binary on different cores we can use the available Barrelfish functions found in `include/barrelfish/spawn_client.h`

```
#include <barrelfish/spawn_client.h>
...
errval_t spawn_program(coreid_t coreid, const char *path, char *const_argv[].
           char *const envp[], spawn_flags_t flags, domainid_t *ret_domainid);

errval_t spawn_program_on_all_cores(bool same_core, const char *path,
            char *const argv[], char *const envp[],
            spawn_flags_t flags, domainid_t *ret_domainid);
```

Listing B.6: Spawning multiple instances of the same program in Barrelfish

NUMA control

Fixing a process to run on a specific core, or NUMA node can be done using the `numactl` function in Linux as following:

```
numactl --physcpubind=1 --membind=1 ./run_the_program param1 param2
```

Listing B.7: Setting up process affinity in Linux

In Barrelfish:

```
#include <skb/skb.h>
...
skb_client_connect();
set_skb_affinity();
```

Listing B.8: Setting up process affinity in Barrelfish

Performance measurements

Performance measurements on Linux using OProfile:

```
sudo opcontrol --reset
sudo opcontrol --separate=cpu --event=RETIRED_INSTRUCTIONS:500000:0:1
sudo opcontrol --start

./run_the_program param1 param2

sudo opcontrol --stop
sudo opcontrol --dump
```

```
sudo opreport event:RETIRED_INSTRUCTIONS > ret_instructions.out
```

Listing B.9: Performance measurements on Linux

Doing the performance measurements on Barrelfish:

```
extern "C"{
  #include <barrelfish_kpi/types.h>
  #include <barrelfish_kpi/dispatcher_shared.h>
  #include <barrelfish/curdispatcher_arch>
  #include <arch/x86/barrelfish/perfmon.h>
  #include <arch/x86/barrelfish_kpi/perfmon_amd.h>
}
...
dispatcher_handle_t my_dispatcher = curdispatcher();
uint64_t event = EVENT_AMD_INSTRUCTIONS_RETIRED;
uint64_t umask = 0;
int counter = 0; // or 2, for the second performance counter
perfmon_setup(my_dispatcher, counter, event, umask, false);
uint64_t start_event_count = rdpmc(0); // rdpmc(1), for the second
...
uint64_t stop_event_count = rdpmc(0); // rdpmc(1), for the second
uint64_t event_count = stop_event_count - start_event_count;
```

Listing B.10: Performance measurements in Barrelfish

Appendix C

List of all performance counter events used

A list of all the performance counter events that were used in the experiments with a short explanation of the event, and the setup in which they were used in the measurements.

TABLE C.1: AMD performance counter events used

Event select	Unit mask	Event abbreviation	Short description
0x76	0x00	cpu_clocks_unhalted	cpu clocks not halted
0xC0	0x00	retired_instructions	retired instructions
0x6C	0x07	system_read	system read responses
0x6D	0x01	system_write	quad-/octwords written to system
0xE0	0x07/0x38	DRAM_accessess	DRAM accesses
0x40	0x00	DC_accesses	data cache accesses
0x42	0x1E	DC_refills_L2	data cache refills from L2
0x43	0x1E	DC_refills_system	data cache refills from NB
0x80	0x00	IC_fetches	instruction cache fetches
0x82	0x00	IC_refills_L2	instruction cache refills from L2
0x83	0x00	IC_refills_system	instruction cache refills from NB
0x7D	0x07	L2_requests	L2 cache requests
0x7E	0x07	L2_misses	L2 cache misses
0x7F	0x03	L2_fill_write	L2 cache fill/writeback
0x4E0	0xF7	L3_requests	L3 cache requests
0x4E1	0xF7	L3_misses	L3 cache misses
0x45	0x00	DTLB_L1M_L2H	L1 DTLB miss and L2 DTLB hit
0x46	0x00	DTLB_L1M_L2M	L1 DTLB and L2 DTLB miss

A list of all the formulas used for the computations:

Instructions per cycle (IPC)

Instructions per cycle is a general measure of computational efficiency. Higher values of the IPC indicate that a more useful work is done in a given amount of time unit.

Events gathered and formulas used

The computation of IPC requires collection of two basic events:

1. `cpu_clock_unhalted`
2. `retired_instructions`

IPC is then calculated with the following formula:

$$\text{IPC} = \frac{\texttt{retired_instructions}}{\texttt{cpu_clock_unhalted}} \tag{C.1}$$

CPI (cycles per instruction) is calculated similarly with:

$$\text{CPI} = \frac{\texttt{cpu_clock_unhalted}}{\texttt{retired_instructions}} \tag{C.2}$$

Memory Bandwidth

These measurements help determine memory bandwidth utilization. Literature suggests that we should apply these measurements especially when an application moves a large amount of data between the memory and the processor.

Experiment theory: Events gathered and formulas used

The computation of memory bandwidth utilization requires collecting the following events:

1. `cpu_clocks_unhalted`
2. `system_read`
3. `system_write`
4. `DRAM_accesses`

Since the AMD SantaRosa processor perform 8-byte write transfers, for calculating the derived measurements we were using the following formulas:

$$\text{Read bytes transferred} = (\texttt{system_read} * \text{period}) * 64 \quad (C.3)$$

$$\text{Write bytes transferred} = (\texttt{system_write} * \text{period}) * 8 \quad (C.4)$$

$$\text{DRAM bytes transferred} = (\texttt{DRAM_accesses} * \text{period}) * 64 \quad (C.5)$$

Furthermore we use the `cpu_clocks_unhalted` to measure the elapsed time. The formula that gives us the elapsed time is the following:

$$\text{Seconds} = \frac{(\texttt{cpu_clocks_unhalted} * \text{period})}{\text{clock frequency}} \quad (C.6)$$

Bandwidth is then calculated as the number of bytes that were transferred in a unit of time, i.e. using the following formula:

$$\text{Read bandwidth}\,(B/s) = \frac{\text{Read bytes transferred}}{\text{Seconds}} \quad (C.7)$$

$$\text{Write bandwidth}\,(B/s) = \frac{\text{Write bytes transferred}}{\text{Seconds}} \quad (C.8)$$

$$\text{DRAM bandwidth}\,(B/s) = \frac{\text{DRAM bytes transferred}}{\text{Seconds}} \quad (C.9)$$

Data cache misses and miss ratio

Good cache behavior is important for good performance. Data caches favor programs that have good spatial and temporal locality for data access. Therefore it is recommended that we always check for data cache behavior.

Events gathered and formulas used

The computation of data cache (DC) miss rate requires collection of the following events:

1. `retired_instructions`
2. `DC_accesses`
3. `DC_refills_L2`
4. `DC_refills_system`.

The formulas that were used are the following:

$$\text{DC request rate} = \frac{\texttt{DC_accesses}}{\texttt{retired_instructions}} \tag{C.10}$$

$$\text{DC miss rate} = \frac{(\texttt{DC_refills_L2} + \texttt{DC_refills_system})}{\texttt{retired_instructions}} \tag{C.11}$$

$$\text{DC miss ratio} = \frac{(\texttt{DC_refills_L2} + \texttt{DC_refills_system})}{\texttt{DC_accesses}} \tag{C.12}$$

Instruction cache misses and miss ratio

The instruction cache contains the most recently fetched x86 instructions and is able to provide them quickly to the pipeline when needed. Having the critical parts of the code fitting in the instruction cache and reusing it is crucial for getting a good performance. As always, low instruction cache miss rate and ratio is desirable.

Events gathered and formulas used

The computation of instruction cache (IC) miss rate requires collection of the following events:

1. `retired_instructions`
2. `IC_fetches`
3. `IC_refills_L2`
4. `IC_refills_system`

The formulas that were used are the following:

$$\text{IC request rate} = \frac{\texttt{IC_accesses}}{\texttt{retired_instructions}} \tag{C.13}$$

$$\text{IC miss rate} = \frac{(\texttt{IC_refills_L2} + \texttt{IC_refills_system})}{\texttt{retired_instructions}} \tag{C.14}$$

$$\text{IC miss ratio} = \frac{(\texttt{IC_refills_L2} + \texttt{IC_refills_system})}{\texttt{IC_accesses}} \tag{C.15}$$

L2 cache misses and miss ratio

Data and instructions are written to L2 when evicted from their respective L1 cache. Having an L2 cache miss will result in either accessing a lower level cache or main memory, both causing a more expensive retrieval of the requested data. Thus it is desirable to have low L2 cache miss rate and ratio, and with it a better spatial and temporal locality for better performance.

Events gathered and formulas used

The computation of L2 cache miss rate requires collection of the following events:

1. `retired_instructions`
2. `L2_requests`
3. `L2_misses`
4. `L2_fill_write`

The formulas that were used are the following:

$$\text{L2 request rate} = \frac{(\texttt{L2_requests} + \texttt{L2_fill_write})}{\texttt{retired_instructions}} \tag{C.16}$$

$$\text{L2 miss rate} = \frac{\texttt{L2_misses}}{\texttt{retired_instructions}} \tag{C.17}$$

$$\text{L2 miss ratio} = \frac{\texttt{L2_misses}}{(\texttt{L2_requests} + \texttt{L2_fill_write})} \tag{C.18}$$

L3 cache misses and miss ratio

The L3 cache is a non-inclusive victim cache holding the cache lines evicted from L2 cache. Good performance requires low L3 cache miss rate and miss ratio.

Events gathered and formulas used

The computation of L3 cache miss rate requires collection of the following events:

1. `retired_instructions`
2. `L3_requests`
3. `L3_misses`

The formulas that were used are the following:

$$\text{L3 request rate} = \frac{\texttt{L3_requests}}{\texttt{retired_instructions}} \tag{C.19}$$

$$\text{L3 miss rate} = \frac{\texttt{L3_misses}}{\texttt{retired_instructions}} \tag{C.20}$$

$$\text{L3 miss ratio} = \frac{\texttt{L3_misses}}{\texttt{L3_requests}} \tag{C.21}$$

Data TLB misses and miss ratio

Translation lookaside buffers (TLB) help the processor with the translation of virtual to physical addresses. They hold the most recently used page mappings to accelerate address translation. TLB behavior favors programs with good spatial and temporal locality.

Events gathered and formulas used

The computation of Data TLB miss rate requires collection of the following events:

1. `retired_instructions`
2. `DC_accesses`
3. `DTLB_L1M_L2H`
4. `DTLB_L1M_L2M`

The formulas that were used for the L1 data TLB measurements are the following:

$$\text{L1 DTLB request rate} = \frac{\texttt{DC_accesses}}{\texttt{retired_instructions}} \tag{C.22}$$

$$\text{L1 DTLB miss rate} = \frac{(\texttt{DTLB_L1M_L2H} + \texttt{DTLB_L1M_L2M})}{\texttt{retired_instructions}} \tag{C.23}$$

$$\text{L1 DTLB miss ratio} = \frac{(\texttt{DTLB_L1M_L2H} + \texttt{DTLB_L1M_L2M})}{\texttt{retired_instructions}} \tag{C.24}$$

Similarly, the formulas that were used for the L2 data TLB measurements are:

$$\text{L2 DTLB request rate} = \frac{(\texttt{DTLB_L1M_L2H} + \texttt{DTLB_L1M_L2M})}{\texttt{retired_instructions}} \tag{C.25}$$

$$\text{L2 DTLB miss rate} = \frac{\texttt{DTLB_L1M_L2M}}{\texttt{retired_instructions}} \tag{C.26}$$

$$\text{L2 DTLB miss ratio} = \frac{\texttt{DTLB_L1M_L2M}}{(\texttt{DTLB_L1M_L2H} + \texttt{DTLB_L1M_L2M})} \tag{C.27}$$

Bibliography

[1] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, MMCS'08, June 2008. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.3289&rep=rep1&type=pdf.

[2] Tudor Salomie, Ionut Subasu, Jana Giceva, and Gustavo Alonso. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *Proceedings of the Eurosys Conference*, April 2011. URL http://eurosys2011.cs.uni-salzburg.at/pdf/eurosys2011-salomie.pdf.

[3] Amadeus SA. http://www.amadeus.com.

[4] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2:706–717, August 2009. ISSN 2150-8097. URL http://portal.acm.org/citation.cfm?id=1687627.1687707.

[5] Gustavo Alonso, Donald Kossmann, and Timothy Roscoe. SwissBox: An architecture for Data Processing Appliances. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, pages 32–37, January 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper4.pdf.

[6] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005. ISBN 1-59593-154-6. URL http://portal.acm.org/citation.cfm?id=1083592.1083658.

[7] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51:77–85, December 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1409360.1409380. URL http://doi.acm.org/10.1145/1409360.1409380.

[8] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13:23–52, March 1988. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/42201.42203. URL http://doi.acm.org/10.1145/42201.42203.

[9] Phillip M. Fernandez. Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms. In *SIGMOD Conference*, page 492, 1994.

[10] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1:188–200, August 2008. ISSN 2150-8097. doi: http://dx.doi.org/10.1145/1453856.1453882. URL http://dx.doi.org/10.1145/1453856.1453882.

[11] Tudor Salomie, Gustavo Alonso, and Donald Kossmann. Shared Scans on Column Stores. Under submission.

[12] 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. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629579. URL http://doi.acm.org/10.1145/1629575.1629579.

[13] 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. *SIGOPS Oper. Syst. Rev.*, 37:90–105, October 2003. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1165389.945455. URL http://doi.acm.org/10.1145/1165389.945455.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29:251–266, December 1995. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/224057.224076. URL http://doi.acm.org/10.1145/224057.224076.

[15] 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 conference on Hot topics in parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1863086.1863096.

[16] 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. *SIGOPS Oper. Syst. Rev.*, 25:95–109, September 1991. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/121133.121151. URL http://doi.acm.org/10.1145/121133.121151.

[17] Barrelfish. Capability Management. Technical report, . https://wiki.netos.ethz.ch/static/barrelfish/doc/TN-013-CapabilityManagement.pdf.

[18] Barrelfish. Inter-dispatcher communication. Technical report, . https://wiki.netos.ethz.ch/static/barrelfish/doc/TN-011-IDC.pdf.

[19] GNU C Library, . http://www.gnu.org/s/libc/.

[20] NICTA. www.nicta.com.au.

[21] GNU Standard C++ Library, . http://gcc.gnu.org/libstdc++/.

[22] LLVM libc++, . http://libcxx.llvm.org/index.html.

[23] CMake. http://www.cmake.org/.

[24] Barrelfish. Hake. Technical report, . https://wiki.netos.ethz.ch/static/barrelfish/doc/TN-003-Hake.pdf.

[25] Boost C++ Libraries. http://www.boost.org/.

[26] Google-sparsehash. http://code.google.com/p/google-sparsehash/.

[27] Georgios Giannikis. Daedalus: A Distributed Crescando System. http://systems.ethz.pubzone.org/servlet/Attachment?attachmentId=126&versionId=1380102.

[28] Paul Drongowski. *Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors*. Advanced Micro Devices, Inc., September 2008. http://ad.amddevcentral.com/Assets/Basic_Performance_Measurements.pdf.

[29] John Levon. *OProfile manual*. OProfile, 2000-2004. http://oprofile.sourceforge.net/doc/index.html.

[30] *BIOS and Kernel Developer's Guide for NPT Family 0Fh Processors*. Advanced Micro Devices, Inc., October 2009. http://support.amd.com/us/Processor_TechDocs/32559.pdf.

[31] STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[32] *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. Advanced Micro Devices, Inc., April 2010. http://support.amd.com/us/Processor_TechDocs/31116.pdf.

[33] Andi Kleen. An NUMA API for Linux. Technical report, August 2004. http://www.halobates.de/numaapi3.pdf.

[34] WikiPage. https://trac.systems.inf.ethz.ch/trac/systems/mcdb/wiki/ClockScanningColumnStoresBarrelfish.