Cloud-ready scalable and elastic data processing using off-the-shelf databases, replication and virtualization

Author(s):
Salomie, Tudor-Ioan

Publication Date:
2014

Permanent Link:
https://doi.org/10.3929/ethz-a-010150199

Rights / License:
In Copyright - Non-Commercial Use Permitted
Cloud-ready scalable and elastic data processing using off-the-shelf databases, replication and virtualization

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

TUDOR-IOAN SALOMIE

Diploma Engineer in Computer Science,
Technical University of Cluj-Napoca, Romania

born on
May 4, 1984

citizen of
Romania

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zürich, Switzerland), examiner
Prof. Dr. Timothy Roscoe (ETH Zürich, Switzerland), co-examiner
Dr. Timothy Harris (Oracle Research Labs, Cambridge, UK), co-examiner

2014
Abstract

Software-as-a-service and cloud computing have gained much traction in recent years. The tightly coupled hardware infrastructure and software offerings of many cloud vendors such as Amazon, Rackspace, or Windows Azure have allowed a significant number of businesses to adopt a pay-as-you-go model and switch from on-premise to cloud architectures.

While many system software solutions can easily benefit from cloud vendor offerings, off-the-shelf relational databases face many challenges, effectively limiting the migration to cloud of entire application stacks. Given the architectures of off-the-shelf databases, problems arise in terms of performance, scalability, and elasticity. These problems are rooted in different places. For example, virtualization technologies, which are the cornerstone of most cloud providers, have a bad reputation for affecting database performance. Elasticity of cloud deployed applications is reduced as existing database engines can not easily take advantage of migration possibilities in virtualized environments. Static peak resource provisioning for databases dramatically reduces the efficiency and the cost-saving benefits promoted through resource sharing and service consolidation in the cloud. Finally, increasingly common multicore servers also pose substantial challenges to database performance and scalability.

Reacting to demands for cloud-ready data processing solutions, many cloud-tailored systems have been developed. While these systems try to replace conventional databases, they differ in functionality, consistency levels, support for queries or transactions, and interfaces. On one hand, such differences are acceptable for certain architectures or newly designed system. On the other hand, there are systems that require the full feature set of relational off-the-shelf databases or for which the redesign of the data processing layer is too expensive. Either way, we argue that efficiently supporting off-the-shelf databases in the cloud is an important problem, which we address in this thesis.

Consequently, we present the design, implementation and evaluation of Vela, a system that combines snapshot isolation replication with virtualization in order to provide a flexible solution for running unmodified off-the-shelf databases, that takes advantage of the elasticity and scalability of the cloud.

Unlike existing replication-only solutions, our system works well both in the case of large servers and in the case of clusters. It achieves this by treating multicore architectures as a distributed system rather than solely relying on the parallel nature of the hardware. We analyzed the limitations of database engines when running on multicore using existing open-source database engines and showed
how to deploy several replicated engines within a single multicore machine to achieve better scalability and stability than a single database engine operating on all cores.

Unlike virtualization-only solutions, our system provides better performance, enhanced functionality and more flexibility in the deployment. It improves flexibility through an online reconfiguration API which can be manually or automatically used to meet service level agreements. By relying on a novel mechanism, called Application Level Ballooning, Vela can take advantage of fine-grained online reconfiguration of the memory footprint of virtualized databases. Finally, based on the results of evaluating the performance of virtualized databases in different setups and under varied workloads we were able to characterize the overheads of virtualization and avoid them through careful engineering and overall system design.
Zusammenfassung


Verglichen mit anderen Lösungen die nur Replikation anbieten, funktioniert unser System auch nachweisbar für große Server und Cluster. Dies wird erreicht, indem es Multi-Core Architekturen als ein verteiltes System behandelt, anstatt sich nur auf die parallelen Eigenschaften der Hardware zu verlassen. Wir haben die Einschränkungen verschiedener Datenbanken auf Multi-Core Systemen anhand verschiedener Open-Source Datenbanken analysiert und zeigen, wie man verschiedene, replizierte Datenbanken auf einer einzigen Multi-Core Maschine als System aufsetzt, welches bessere Skalierbarkeit und Stabilität aufweisen kann als eine einzige Datenbank die auf allen Cores operiert.

Acknowledgments

The foremost recognition goes to my advisor, Gustavo Alonso, for his guidance, advice and enthusiasm in all my endeavors. He has been a true mentor and role model for me, inspiring me both in my research and personal activities. From him I have learned much about critical thinking and thoroughness in research. I will always admire his leadership, intuition and way of thinking.

My thanks extend to Timothy Roscoe, Donald Kossmann and Kevin Elphinstone who have played important roles in guiding my research. They enlightened me many times and taught me what questions to ask in my research. I also want to thank Timothy Harris for accepting to be part of my defense committee and for shepherding my work on Application Level Balloning.

Jana, Ionut and Ilias have added much value to the Vela system, making coding and benchmarking a team effort. Working with you was very productive and interesting for me. I thank you all for your contributions.

My colleagues Akhi, Alex, Cagri, Darko, Ercan, Gerd, Gerogios, Kornilios, Louis, Markus, Martin, Philipp, Prataniu, Pravin, Qin, Simon L., Simon G., Stefan K., and Zsolt have made tackling hard challenges much easier and savoring victories so much fun. My special thanks go to the great administrative staff of the Systems Group that was ever so friendly and helpful: Eva, Jena, Simonetta and Karel. To all the present and past colleagues in the Systems Group I want to extend my sincere gratitude for our fruitful discussions. Also, the times we shared in and out of the office will not be forgotten.

When I needed a break from coding, debugging, or writing, my friends Iulia, Ioana F., Andrei, Laurentiu, Bogdan, Radu, Hoartiu, and Roberto were there for me.

I am forever grateful to my parents for supporting me in all my choices and for bearing the distance at which these have kept us.

Finally, I thank you, Ioana, for being by my side, rejoicing victories, putting up with me during difficult times, and giving me inspiration when most needed.

To you, the reader, I thank you for your interest in my work.
# Contents

1 Introduction .......................................................... 1
  1.1 Solution overview ............................................. 2
  1.2 Contributions .................................................. 6
  1.3 Structure ....................................................... 8

2 Motivation and background .......................................... 9
  2.1 The divergence of data processing solutions ................. 9
  2.2 The case for off-the-shelf databases ....................... 12
  2.3 The case for replication ...................................... 13
  2.4 Virtualization is not enough ................................ 15
  2.5 Scaling data processing on multicores .................... 19
    2.5.1 Off-the-shelf database scalability on multicores ...... 21
    2.5.2 Replication within the same machine ................. 26
  2.6 Dynamic reconfiguration .................................... 27
    2.6.1 Coarse grained reconfiguration ...................... 27
    2.6.2 Fine grained reconfiguration ......................... 28
  2.7 Chapter summary ............................................... 33

3 Vela: design and implementation .................................. 35
  3.1 Design goals .................................................. 36
  3.2 Data replication .............................................. 36
  3.3 System components ........................................... 40
    3.3.1 The router ................................................ 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2 The data processing instances</td>
<td>43</td>
</tr>
<tr>
<td>3.3.3 The system model</td>
<td>46</td>
</tr>
<tr>
<td>3.4 Virtualization</td>
<td>48</td>
</tr>
<tr>
<td>3.5 Dynamic reconfiguration API</td>
<td>50</td>
</tr>
<tr>
<td>3.6 Target functions</td>
<td>52</td>
</tr>
<tr>
<td>3.6.1 Automatic satellite scaling</td>
<td>53</td>
</tr>
<tr>
<td>3.7 System deployment examples</td>
<td>54</td>
</tr>
<tr>
<td>3.7.1 Single machine deployments</td>
<td>54</td>
</tr>
<tr>
<td>3.7.2 Cluster deployments</td>
<td>58</td>
</tr>
<tr>
<td>3.7.3 Amazon EC2 deployments</td>
<td>59</td>
</tr>
<tr>
<td>3.8 Chapter summary</td>
<td>59</td>
</tr>
<tr>
<td>4 Vela: system evaluation</td>
<td>61</td>
</tr>
<tr>
<td>4.1 Benchmarks overview</td>
<td>62</td>
</tr>
<tr>
<td>4.1.1 TPC-W</td>
<td>62</td>
</tr>
<tr>
<td>4.1.2 TPC-E</td>
<td>63</td>
</tr>
<tr>
<td>4.2 Scaling up over bare metal</td>
<td>63</td>
</tr>
<tr>
<td>4.2.1 Experimental setup, workloads and datasets</td>
<td>64</td>
</tr>
<tr>
<td>4.2.2 PostgreSQL: standalone vs. Vela</td>
<td>65</td>
</tr>
<tr>
<td>4.2.3 MySQL: standalone vs. Vela</td>
<td>69</td>
</tr>
<tr>
<td>4.2.4 Replication overhead</td>
<td>73</td>
</tr>
<tr>
<td>4.2.5 Effects of load separation</td>
<td>73</td>
</tr>
<tr>
<td>4.2.6 Standalone database improvements</td>
<td>74</td>
</tr>
<tr>
<td>4.3 Scaling up with virtualization</td>
<td>77</td>
</tr>
<tr>
<td>4.4 Scaling out over cluster of multicores</td>
<td>81</td>
</tr>
<tr>
<td>4.5 Online system reconfiguration</td>
<td>85</td>
</tr>
<tr>
<td>4.5.1 Manual reconfiguration</td>
<td>86</td>
</tr>
<tr>
<td>4.5.2 Target functions</td>
<td>87</td>
</tr>
<tr>
<td>4.6 Multitenancy support</td>
<td>94</td>
</tr>
<tr>
<td>4.7 Chapter summary</td>
<td>97</td>
</tr>
</tbody>
</table>
5 Application Level Ballooning 99
5.1 Overview and background 100
  5.1.1 Reconfiguring memory allocation 100
  5.1.2 Requirements for ALB 102
5.2 ALB design 104
  5.2.1 Memory ballooning in Xen 105
  5.2.2 Application ballooning on top of Xen 106
  5.2.3 The MySQL ALB module 108
  5.2.4 The OpenJDK ALB module 109
  5.2.5 Changes to the Linux kernel 113
  5.2.6 Changes to the Xen balloon driver 113
5.3 Experimental evaluation 115
  5.3.1 Experimental setup 116
  5.3.2 Overhead of ballooning 116
  5.3.3 In-flight memory resizing 120
  5.3.4 An end-to-end example 123
  5.3.5 Collocating database servers 124
  5.3.6 ALB performance 126
5.4 Implementing ALB 129
5.5 Discussion 130
5.6 Chapter summary 133

6 Databases in virtualized environments 135
6.1 Overview and related work 135
6.2 Experimental setup 136
6.3 Micro benchmarks 140
  6.3.1 I/O write performance 140
  6.3.2 I/O read performance 143
  6.3.3 Memory performance 144
  6.3.4 Network performance 146
6.3.5 Bonding virtual network interfaces ..................................... 149
6.4 Virtualized database performance ........................................... 152
  6.4.1 Main memory storage ..................................................... 152
  6.4.2 Local disk storage ....................................................... 155
  6.4.3 Network storage ........................................................ 158
6.5 Chapter summary .............................................................. 160

7 Conclusion .............................................................. 163
With a growing popularity of cloud deployed applications, an increasing number of cloud based data processing solutions have emerged. Aligning with the cloud promoted properties of scalability and elasticity, new data management solutions were quick to abandon some of the models of the traditional relational database management systems.

The NoSQL revolution yielded systems that introduce trade-offs between functionality and ease of scalability and elasticity. Most of the times these translate into simplified data manipulation APIs (e.g., missing full SQL support), relaxed consistency models (e.g., eventual consistency), or lack of support for data processing operators (e.g., join operators).

On one hand, such solutions are very well suited for many newly developed cloud applications. For example, Foursquaure’s venue check-in store uses MongoDB, IMDb’s 10 star rating system uses DynamoDB, and Netflix customer viewing history is stored using Cassandra. Such systems either do not require features like a rich query API or strong consistency guarantees. When required, the developers implement such functionalities in the application logic above the data storage layer.

On the other hand, migrating applications that rely on traditional relational database management systems to the cloud raises a series of problems. Adhering to the NoSQL movement would require re-writing the applications such that they
can use the new (and sometimes limited) querying APIs and rethinking basic assumptions of the transactional properties, like consistency. The other path is that of naively porting existing data processing solutions to the cloud. While very appealing, such an approach faces a series of problems. Relational data management systems (RDBMS) are designed to be deployed in isolation. We refer to RDBMS solutions that require little to no modification except for configuration and performance tuning and which are designed to be generally suitable for a large spectrum of workloads as off-the-shelf databases. Examples of off-the-shelf databases are: MySQL, PostgreSQL, IBM DB2, Oracle, etc. This class of databases are designed to be provisioned with hardware resources for peak workloads, have limited support for online reconfiguration, which in turn leads to a reduced ability to scale on demand. We also point out that most cloud infrastructures rely on virtualization for abstracting the hardware. While virtualization brings many advantages to cloud infrastructures, it does not always interact nicely with off-the-shelf databases.

In this thesis, we present work in the context of enabling off-the-shelf databases to be efficiently deployed and used at scale in large clusters and cloud infrastructures. After further motivating the need for running off-the-shelf databases in cloud infrastructures, we present the current functional and performance limitations of off-the-shelf database in virtualized environments. We continue by addressing these problems in the design and implementation of a system that relies on off-the-shelf databases while also adhering to cloud properties: it is scalable, elastic and can be dynamically and automatically reconfigured, while supporting multitenancy.

1.1 Solution overview

In order present an overview of our work, we refer the reader to Figure 1.1. Vela is the solution we present in this thesis for efficiently deploying, administering and running off-the-shelf databases on cloud infrastructures. Vela is based both on existing technologies like data replication and virtualization, and on novel techniques like those that we developed in the Multimed system, or mechanisms like Application Level Ballooning (ALB).

Vela combines snapshot isolation replication with virtualization. Unlike replication-only solutions, Vela works well both in the case of large servers, and in the case of clusters. Unlike virtualization only solutions, our system provides better performance and more flexibility in the deployment.
1.1. SOLUTION OVERVIEW

Multimed

Vela’s ability to work well within large servers is inherited from Multimed, a system that we designed for scaling off-the-shelf databases on large multicore machines.

In spite of the intense research in the area, there are still few practical solutions that can efficiently scale with the increasing cores and memory found in new multicore servers and that allow a flexible deployment of databases on such machines. The cause of these are the substantial challenges coming from the increase in cores and non-uniform memory access.

As databases face structural limitations, we developed a novel solution to the scalability problem of off-the-shelf databases. Our approach treats a large multicore machine as a set of distributed resources and scales data processing on such machines through distributed techniques like replication rather than parallelization. By deploying several replicated engines within a single multicore machine we show how to achieve better scalability and stability than a single database engine operating on all cores. To the best of our knowledge, this is the first approach of its kind to this problem.

We implemented Multimed, a system based on data replication which does not require any costly re-engineering of off-the-shelf databases and works well in terms of scalability and performance for a wide range of uses cases, supporting multiple off-the-shelf database engines, like PostgreSQL or MySQL.

Multimed uses a primary copy approach (the master database) running on a subset of the cores. The master database receives all the update load and propagates the changes to satellite databases. The satellites store copies of the database and

![Figure 1.1: Context overview of the presented solution](image-url)
run on non overlapping subsets of the cores. These satellites are kept in sync with the master copy and are used to execute the read only load. The system guarantees global consistency in the form of snapshot isolation, although alternative consistency guarantees are possible.

Our experiments show that a minimally optimized version of Multimed exhibits higher throughput with lower response times and more stable behavior as the number of cores increase than standalone versions of off-the-shelf databases for standard benchmark loads.

We consider that Multimed is representative of a new architecture for database systems in the context of multicores relying on well known distributed system techniques and proving to be widely applicable for many workloads.

Like any database, Multimed is not suitable for all possible use cases but it does support a wide range of scenarios. For example, in the case of the TPC-W benchmark \[121\], Multimed can support all update rates, from the browsing and shopping mix to the ordering mix, with only a slight performance degradation for the ordering mix. For business intelligence and data warehouse loads, Multimed can offer linear scalability by assigning more satellites to the analytical queries.

**Vela**

Following a natural evolution from Multimed, Vela extends the scalability from within a single machine to a cluster of servers through the same data replication approach.

One of the main goals of Vela is to scale data processing using off-the-shelf databases over clusters of multicores. The system relies on virtualization (to take advantage of the system architectural possibilities available in the cloud) and the same snapshot isolation replication model, as in Multimed (for fault tolerance, performance, and scalability, without sacrificing consistency).

Replication has become a de facto standard for achieving scalability and availability with industry solutions like SQL Azure [22], Google’s Megastore [6], or Oracle RAC [92] relying on it. It is also frequently used in research prototypes like the Tashkent [42], Postgres-R [68] or Ganymed [99].

Having a unified scalability model, both for scaling up (within a machine) and for scaling out (over a cluster), the Vela system can seamlessly react to both the number and the size of the machines it spans over. Scaling up refers its ability to efficiently utilize the resources of a single machine and to yield performance gains as more resources (in our case CPU and memory) are added. Scaling out refers to the
1.1. SOLUTION OVERVIEW

ability to utilize and gain performance benefits from adding more machines (cluster nodes) to the system. In most replication based systems, scaling out and up are seen as two orthogonal problems as there is no uniform view of resources within a machine and in the cluster. Using virtualization as an abstraction layer over clusters of multicore machines, Vela has a uniform view of the available resources. Vela couples virtualization with replication for scaling both up and out.

Through virtualization, Vela uses techniques such as memory ballooning [7, 115], hot plugging/removing cores or live-migration [30] to add flexibility and support for online reconfiguration.

The performance evaluation of the system, relying on standard benchmarks, shows that it scales both on large servers and on clusters, it supports multi-tenancy and can be dynamically reconfigured at run-time, either automatically or manually. In scalability tests, spanning dozens of nodes, the system handles more than 15,000 transactions per second, servicing load for 400 clients, while maintaining response times less than 50 milliseconds. The multi-tenancy experiments show that the system can easily accommodate 20 collocated tenants while maintaining response times below 400 milliseconds under an aggregated load generated by 800 clients. This shows the system is practical for a modern workloads, scaling up and accommodating high levels of concurrency and multi-tenancy.

Application Level Ballooning (ALB)

Using virtualization in conjunction with off-the-shelf databases implies a tradeoff between the uniform view of resources and possible performance overheads and functional miss-matches of databases and virtualization capabilities. Many of the performance overheads are avoided in Vela through its design and careful engineering. In order to address some of the functional miss-matches, we have designed and implemented Application Level Ballooning (ALB).

Virtualization in cloud computing and server consolidation enables applications previously deployed on dedicated machines to share a physical server, reducing resource consumption, energy, and space costs among other benefits. Multiplexing such servers and ensuring application performance as load changes, however, requires careful coordination policies, and virtual machine monitor (VMM) mechanisms like live migration and memory ballooning. Such mechanisms are used to dynamically reallocate resources (e.g., main memory) in virtual machines (VMs) to meet performance goals. These techniques work well when the OS manages application memory via paging. However, they do not work well when the application manages memory itself – the common case in server applications like
databases and language runtimes. Here, efficient execution depends on the program having an accurate picture of available resident memory. Reallocating RAM on these systems using standard ballooning severely impacts performance, leading to thrashing and, in some cases, failure.

ALB, is a technique for reallocating RAM among a collection of applications that manage their own memory running in collocated VMs. ALB re-allocates RAM without shutting applications down or reconfiguring them. It is fast, effective, requires minimal changes to applications, and crucially preserves the ability of an application to optimize performance based on having an accurate idea of the available physical RAM.

1.2 Contributions

In designing, implementing and evaluating our solution, the following contributions have been made:

**Multimed** The main contribution of Multimed is to show that a shared-nothing design, similar to that used in clusters, works well in multicore machines, surpassing the scalability and performance of standalone off-the-shelf databases. The big advantage of such an approach is that the database engine does not need to be modified to run in a multicore machine. The parallelism offered by multicore is exploited through the combined performance of a collection of unmodified engines rather than through the optimization of a single engine modified to run on multiple cores. An interesting aspect of Multimed is that each engine is restricted in the number of cores and the amount of memory it can use. Yet, the combined performance of several engines is higher than that of a single engine using all the cores and all the available memory.

**Vela** The contribution of Vela is in showing how to seamlessly scale data processing both up on large multicore machines and out over clusters, using off-the-shelf database engines and a single data replication model. Although based on known techniques like replication and virtualization, the design of Vela faced several non-trivial challenges. The biggest ones were in designing the system such that it can seamlessly take advantage of both the size and the number of cluster machines over which it is deployed. First, handling the two dimensions (of size and number) separately makes reasoning and taking decisions (e.g., deployment, online reconfiguration) about the system more difficult. Virtualizing the hardware reduces the complexity with a minimal cost in resources and performance. Further, having a homogeneous view of
the machines allowed us to define an API for the online reconfiguration of the system. Second, managing a dynamic set of replicated off-the-shelf database instances over a virtualized cluster required a coordinator component that implements the replication logic, monitors the deployment, and automatically takes decisions in order to react to load changes. Third, addressing performance and functional miss-matches between databases and virtual machine monitors required engineering effort in order to achieve good scalability and support the reconfiguration API.

Application Level Ballooning As systems software like databases and language runtimes typically manage memory themselves (exploiting application knowledge unavailable to the OS) and have static peak memory provisioning, they are ill-suited for service collocation and online reconfiguration in virtualized environments. The only existing options for dynamically reallocating physical memory between VMs badly impact the performance of applications which manage their own memory. Our contribution is a novel novel technique for adding memory elasticity to such applications. Our approach address this problem by extending Virtual Machine memory ballooning to applications so that memory can be efficiently and effectively moved between virtualized instances as the demands of each change over time. The results are significantly lower memory requirements to provide the same performance guarantees to a collocated set of Virtual Machines running such applications, with minimal overhead or intrusive changes to application code.

The contributions presented in this thesis have also been reported in the following publications:


CHAPTER 1. INTRODUCTION


1.3 Structure

The following Chapter 2 further motivates the need for- and advantages of running off-the-shelf databases over large clusters and cloud-like infrastructures, relying on replication and virtualization. Chapter 3 presents the design and implementation details of Multimed and Vela, while Chapter 4 presents the experimental evaluation, focusing on scalability, manual and automatic dynamic reconfiguration, and multi-tenancy support. Chapter 5 presents the design, implementation and evaluation of the Application Level Ballooning mechanism used in connection with Vela’s reconfiguration API. In Chapter 6 we investigate the performance overheads of virtualized off-the-shelf databases under different workloads and setups. Chapter 7 concludes the dissertation and presents final thoughts and research avenues worth pursuing in the future.
This chapter establishes the context and motivates the work presented in the thesis. It compares current approaches to data processing in cloud environments, it motivates the usefulness of our approach by discussing the advantages and trade-offs of running off-the-shelf databases over large clusters and cloud setups, and it presents the techniques underlying our approach, such as replication for achieving scalability and virtualization for flexibility. For all topics, we cover advantages, drawbacks, and ways to mitigate them.

2.1 The divergence of data processing solutions

As argued on several occasions by Stonbraker et al. [112, 114], there is an increasing divergence in approaches to data processing. Historically, generic database management have been built, which provided single solutions to different data processing problems. As we show in this section, this has changed over time as requirements and hardware have evolved.

In the early 1980s, DBMSes have focused on addressing topics related to Online Transaction Processing (OLTP). Initial prototypes (e.g., System R, INGRES) evolved to commercial solutions (Oracle, DB2, etc.) that supported ACID transactions, optimized disk-based storage data structures (e.g., B-Trees) and developed techniques for mitigating disk I/O latencies [48].
CHAPTER 2. MOTIVATION AND BACKGROUND

With increasing amounts of data being stored and processed in different business areas, companies wanted to better understand and process their data. In the mid 1990s, OLTP remains important, but support for Online Analytical Processing (OLAP) workloads becomes a commonly requested feature of databases. The data over which such workloads is run is now centralized in “data warehouses”. Compared to OLTP, OLAP is not characterized by fast update transactions, but rather by complex ad-hoc queries that often rely on heavy aggregation. The size of OLAP data is comparably larger to that of OLTP, the workload is no longer update intensive, and the requests are not exclusively transactional but rather shifted towards complex ad-hoc queries. The divergence between OLTP and OLAP translated to new data schemas, new data structures, and new query optimizers.

Consequently, database management systems evolved, trying to support both types of datasets and workloads. While single products provided support for OLTP and OLAP, a clear differentiation was done internally, in the database, between the two. Most notably, this is visible in the fact that OLTP and OLAP are not concurrently supported on the same data.

With larger adoption of the web, the demanded scale of data processing increases and new business requirements are formulated (e.g., support for higher throughput rates, lower latencies). Consequently, databases evolved outside of a single machine to clusters. This lead to new distributed database models based on data partitioning and replication [26, 48, 47].

In a next step cloud computing induced a redesign of data processing systems, which aligned to cloud characteristics like elasticity, availability or fault-tolerance [19]. The new class of systems, called NoSQL, achieved these properties by giving up old-established database properties like consistency, rich schemas, or support for complex queries.

Targeting specialized workloads in financial services (algorithmic trading) or large scale data analytics – alternative data processing models gained popularity. For example, the Map-Reduce [38] programming model which works in steps of partitioning and distributing data for processing (map) and then aggregating the intermediary results (reduce) influenced warehouse data processing systems. Though criticized for frequent use in place of existing database management systems [113], Apache Hive, Pig [90], HBase or BigTable [27] offer data management solutions tailored for highly parallel workloads over large datasets. Another example are Event Stream Processing (ESP) [1] systems which track continuous streams of data and match them with fixed sets of queries, trying to identify useful events in the data stream as fast as possible. Ideas for ESP have also influenced traditional database management systems like C-Join [25] (a scalable join operator for high concurrency data warehouses), Crescando [124] (a scalable and predictable
2.1. THE DIVERGENCE OF DATA PROCESSING SOLUTIONS

scan-sharing relational table implementation) or ShareDB [44] (a new database architecture based on query batching and shared computation).

Finally, hardware changes of the past decade forced database engines to specialize even more. The memory-wall [13] in which the CPU frequencies have far out-grown the memory bandwidth brought along different data representation models. Large amounts of RAM in servers allow entire datasets to be main-memory resident, requiring new approaches to data processing. The Decomposition Storage Model [134] (DSM, or column-storage) proves much better for OLAP workloads, while the N-ary Storage Model (NSM, or row-storage) is better suited for OLTP workloads. The increase in the number of cores in a machine and the presence of Cache-Coherent Non-Uniform Memory Access hardware protocols also impact databases: while the increased degree of parallelism can be useful for certain workloads, it mostly causes scalability issues in existing database designs [116, 109].

Looking back, the trend is clear: data scale, hardware and business requirements are diversifying. A single system that addresses all these requirements is less and less probable to exist. Consequently, data processing systems need to address specific areas by defining the scale, hardware and business requirements that they are trying to fulfill. The work presented in this thesis positions itself as follows:

Workloads The workloads addressed by Vela are transactional, highly concurrent and read-intensive. Such workloads are characteristic for many online systems and are represented by well-established OLTP benchmarks such as TPC-E [118] or TPC-W [121].

Scale The system should be able to handle large amounts of concurrent requests while minimizing latency, as well as many different customers (degree of multi-tenancy). As the number of concurrent requests increases, the system should be able to react accordingly in order to maintain minimal latencies. As more customer databases are added to the system, they should be able to be efficiently collocated. The deployment scale of the system ranges from clusters for each tenant to large cloud infrastructures.

Hardware The targeted hardware is modern multicore servers. While many off-the-shelf database management solutions have problems in scaling on such hardware, our approach overcomes the core scalability issue. Our approach to scaling off-the-shelf databases relies on replication and heavily benefits from storing replicas in main-memory. Even for datasets in the order of tens of gigabytes, modern servers and cluster infrastructures provide sufficient resource to hold multiple replicas entirely in RAM on each machine.
CHAPTER 2. MOTIVATION AND BACKGROUND

Requirements The solution need to handle transactional, rich and ad-hoc queries expressed using SQL, while providing consistency. Besides these, the system should be elastic so that it can react to the offered load in order to meet SLAs for throughput or response time.

2.2 The case for off-the-shelf databases

A main argument in favor of using off-the-shelf databases in cloud setups comes from the shortcomings of existing cloud-ready solutions.

Cloud-ready solutions offer scalability and elasticity at the cost of reduced functionality, varied levels of consistency, or limited support in queries and transactions. For instance, Dynamo sacrifices consistency under certain scenarios and has a simple put/get query API. Bigtable also has a simple query API that does not support a full relational model. G-Store uses a grouping protocol for transactional access to groups of keys, pushing the logic of managing these groups to the client application. Megastore blends NoSQL and relational DBMSes providing strong consistency and high availability, though only over partitions of data, which requires join operations to be executed in the client application.

There are also counter arguments for running off-the-shelf databases in the cloud. Key to many cloud systems are among others virtualization, elasticity, and online reconfigurability. Off-the-shelf databases are usually deployed on dedicated servers and statically provisioned with resources (CPU, memory) for peak loads. Making assumptions on these available resources, off-the-shelf databases prove not be elastic.

In virtualized environments, hot plugging and removing of resources is a common mechanism for online reconfiguration, supported by most hypervisors like Xen or VMWare. While off-the-shelf databases, both commercial (SystemX – a well known relational database management system) and open source (PostgreSQL, MySQL) will react to hot plugging or removing of CPUs, they face large challenges with memory. This comes from the fact that databases take memory management under their own control (to different degrees). For example at one extreme SystemX is statically configured with a fixed amount of memory that it uses for data caching or for a pool used per-connection for operations like sorting or aggregation. At the other extreme we find PostgreSQL which liberally only takes a small amount of memory (i.e., the “shared_buffer”, recommended $\frac{RAM}{4}$) under its control, relying on the OS disk cache for the rest of the data caching. In between the two we have MySQL (with InnoDB – its most popular storage engine) which takes the memory management of the data heap, locks, etc. under its
control while per-connection memory (sorts, aggregations, etc.) is still managed through the OS. For off-the-shelf systems like PostgreSQL, dynamic memory plugging or removing can be efficiently used, while for systems like MySQL/InnoDB or SystemX, Application Level Ballooning [78, 115] mechanisms are needed. We present our design and implementation of Application Level Ballooning [115] in detail in Chapter 5.

While off-the-shelf databases provide functionality lacking in readily available cloud systems, a naive solution does not work out of the box. We show however that with the design of Vela, scalability, elasticity, collocation and online reconfiguration can be achieved without sacrificing the strong features of off-the-shelf databases. Existing applications that rely on off-the-shelf databases can take full advantage of Vela, without requiring code-rewrite, as Vela does not sacrifice functionality.

2.3 The case for replication

Two are two large classes of data replication models: primary-master and multi-master data replication. In primary-master replication all the data changes are handled by a single database, while in multi-master changes can be processed by any database replica. While primary-master approaches are not be able to handle high update workloads (since all are processed by a single instance) it does not have to deal with conflicting updates since all are handled by a single database engine. In the cast of multi-master replication, where updates are balanced over all replicas, high-update rates can be tolerated (as long as they are not heavily conflicting). The downside of multi-master replication comes from conflict resolution among the masters. We differentiate two types of multi-master systems: synchronous (or eager) which try to prevent conflicts and asynchronous (or lazy) which try to detect and resolve conflicts.

Through data replication systems can gain features like high availability, load balancing, fault tolerance or performance. Postgres-R [68] relies on multi-master eager replication for scaling out update heavy workloads, emphasizing that through careful optimization of communication overheads performance and scalability can be achieved. The Ganymed [99] system uses a primary-master lazy replication model, also for scaling out, though with an emphasis on read-intensive workloads, showing the benefits load balancing over replicas. MySQL Cluster [85] is another multi-master eager replication systems based on MySQL instances that is optimized for write-intensive workloads using replication as a means of achieving high-availability and performance. Garcia et al. [43] and the Heterogeneous Repli-
cated Database system (HRDB) [125] use database replication for addressing the problem of Byzantine faults in databases.

One highly popular data replication model is that based on snapshot isolation. The interest in snapshot isolation based replication [10] comes from its performance benefits along with resilience to concurrency anomalies. Snapshot isolation does not allow dirty reads, non-repeatable reads, phantoms or lost updates. It does however suffer from write-skew which makes Snapshot Isolation weaker than serializability. The write-skew can happen when a transaction $T_a$ reads a value $v_1$ and $v_2$, then another transaction $T_b$ reads $v_1$ and $v_2$, writes $v_2$, and commits. Now when $T_a$ writes $v_1$ the write-skew occurs. Under serializability this could not happen.

With snapshot isolation, a transaction’s reads are consistent within a snapshot of the database at the time the transaction started. Strictly speaking, the snapshot is taken on the transaction start (i.e., when the \textit{BEGIN TRANSACTION} request is sent to the database). In general, if an explicit \textit{BEGIN TRANSACTION} is not issued, then it is implied on the execution of the first SQL SELECT, INSERT, UPDATE or DELETE in the scope of a new transaction. A new transaction can start, for example (but not limited to), after the commit/abort of a previous transaction on the same connection or when a new connection is established.

With snapshot isolation, a transaction can successfully commit if no updates that it has made are in conflict with updates made by a concurrent transaction. The performance benefit of snapshot isolation is clear: all read transactions will succeed and will not be interfered by the write transactions.

Data replication based on snapshot isolation is commonly used in commercial products (like SQL Azure [22] built on top of Cloud SQL Server [11], Teradata [132], or Heroku Postgres [57]) and research prototypes (like Tashkent [42], Ganymed [100], or the one presented by Kemme et al. [74]). All these systems use replication either for scaling the data processing out over clusters or for increasing the system’s availability. What they do not offer is dynamic reconfiguration and flexibility. Also, all these systems are limited by their underlying database engine’s ability to scale up on modern multicore hardware.

The commercial database-as-a-service solution offered by Heroku, called Heroku Postgres, uses an asynchronous replication scheme based on PostgreSQL. The system relies on creating identical copies of a database (they refer to this process as “forking”) that are replicated over a set of machines and are asynchronously receiving updates from the main data. The replicas that handle only read-only workloads are called “followers”. The strong feature of the system is its ease of scaling read-intensive workloads. The downsides of the approach are the limited ability to handle large multicores and a limited description of how it is deployed.
2.4 VIRTUALIZATION IS NOT ENOUGH

and tuned to operate in the virtualized environment of Amazon EC2, which it uses as infrastructure. The system is also static in nature, not supporting online reconfiguration.

Other approaches for database-as-a-service rely primarily on replication and data-partitioning for scaling out processing in cloud-infrastructures. Schism [34] monitors transactional workloads in order to define data placements that minimize the number of distributed transactions. The captured transactional workload is modeled as a graph which is then partitioned such that the occurrence of distributed transactions is minimized. The approach is very fine grained and poses a series of challenges. The SWORD [102] system also described an approach to scaling out transactional data processing for cloud infrastructures based on data partitioning. In order to achieve scalability it relies on a hypergraph partitioning method for minimizing the occurrence of distributed transactions and addresses limitations of the Schism system.

The main challenges in data-partitioning approaches are identifying the workload characteristics, partitioning the data such that distributed transactions are minimized and correctly re-partitioning as a result of workload changes. Such research avenues are not in the scope of the system presented in this thesis. Nonetheless, we will show that the design of Vela supports data replication coupled with data partitioning. As such, approaches like those of Schism or SWORD can nicely complement the work presented in this thesis.

2.4 Virtualization is not enough

In the previous sections of this chapter we have compared different types of data processing solutions and made a case for relational off-the-shelf database engines, as well as for replication as a mechanism for scalability and performance. In this section, we turn our attention to the impact of virtualization (a prominent technology in cloud infrastructures) on relational off-the-shelf databases.

In the context of running databases in virtualized environments favorable arguments emphasize consolidation opportunities and dynamic reconfiguration of resource allocation. We make the argument that we need to clearly identify the drawbacks of virtualizing databases in order to be able to side-step them in Vela’s design.

Recent work indicates a large adoption of virtualized databases in the case of multitenancy, looking at the aspects of tuning database performance by controlling the encapsulating virtual machine [110] or optimal data placement in the storage layer based on the tenant workload characteristics [93].
In order to take advantage of virtualized cloud infrastructures, many storage management and advisory systems have been designed. Naively deploying virtual disks in data center storage arrays can affect overall performance. BASIL [49] is a step forward for storage management and load balancing for virtual disks across storage arrays. The assumption that I/O latency is a first class metric for initial storage placement and subsequent load balancing and migration allows BASIL to significantly decrease latency in many micro-benchmarks as well as to lower the variance of latency for enterprise applications. What is still missing is an analysis of the impact and cost of live migration in data centers. Pesto [50] extends the work presented in BASIL. Through a light-weight online sampling mechanism, Pesto can take into account the device throughput, latency, and outstanding I/O operations for improving the placement of virtual disks in data centers. Compared to BASIL, Pesto takes an active modeling approach through online workload injection for generating workload models. Still, Pesto is used as a storage placement and capacity advisor and does not take into account the actual runtime costs of re-configuring the storage. Romano [96] is another storage management system that addresses the problem of storage consolidation in virtualized data centers. Romano addresses some of the problems of BASIL and Pesto in terms of latency predictions and bad virtual disk placement recommendations. It achieves this by determining its own confidence intervals using prediction intervals. Romano comprises three components: a performance model (storage devices and workloads), an aggregation model and a load balancer.

Virtualizing off-the-shelf databases may lead to a degradation in performance. Several studies investigate this [16], corroborating our experience in building Vela, which we detail in Chapter 6. For most cases, main memory workloads behave very similarly in Native (Bare metal) vs. Virtualized setups, mostly due to Intel’s VT-x and AMD’s AMD-V support for virtualization. For I/O intensive workloads, virtualized performance degrades and CPU utilization increases, as compared to that in Native cases.

Figures 2.1 and 2.2 exemplify this. We compare the performance of PostgreSQL and MySQL running two different workloads in three different setups. The setups are Native (the database engines run on top of Linux 3.6.6, un-virtualized) and two virtualized setups based on Xen (XenPV represents a paravirtualized Virtual Machine and XenPVHVM represents a hardware virtualized Virtual Machine with paravirtualized drivers for block and network devices). The Virtual Machines have 16 cores and 128GB of RAM. For the Native setups, the databases were confined to the same cores (16 in total) and the same NUMA nodes (total of 128GB RAM) as those allocated to the Virtual Machines. We used two different datasets and workloads (based on standard TPC [122] benchmarks) so that we can emphasize different database performance aspects. These experiments were carried out on a
2.4. VIRTUALIZATION IS NOT ENOUGH

64 core AMD Opteron (each core clocked at 2.4GHz), with 512GB of main memory and 8 NUMA nodes. A detailed description of the hardware is given in Section 6.2.

Figure 2.1: Virtualized database performance overhead for non-durable main-memory resident datasets

Figure 2.1 compares PostgreSQL and MySQL in the three setups using a TPC-W based benchmark. The Browsing mix of the benchmark defines a read-intensive workload (95% read-only transactions). The dataset we used is \( \approx 30\text{GB} \) and is completely stored in main-memory. The transaction logs of the databases are also in main-memory. We chose this workload and data placement strategy to put pressure on the transaction manager, concurrent data access, and relational operator executions. The experiment does not put pressure on the transaction log or on the durable storage (i.e., disk) read/write.

The experiment shows that for workloads that are only operating on main-memory there is no performance difference (for PostgreSQL and MySQL) between the non-virtualized and virtualized setups.
Figure 2.2, like in the previous experiment, compares PostgreSQL and MySQL in the three setups, this time using the TPC-C benchmark. The workload consists of 5 transactions that have a much lower response time (compared to the those of the TPC-W workload). The dataset used is $\approx 15$GB. The dataset is cached in main-memory for all setups, but the transaction logs are stored on a local disk. The workload chosen for this experiment was picked so that it emphasize the performance of the database’s transaction log, directly tied into the synchronous write performance to durable storage (i.e., disk).

Both the throughput and the response time plots of PostgreSQL and MySQL indicate that the durable storage I/O overhead of synchronously writing to the transaction log after each completed transaction causes a performance overhead in the case of virtualization. An interesting aspect is that while the performance of both virtualized setups is lower than that of the Native setup, the “shape” and trend of both the response time and the throughput are the same. This shows that
while a latency overhead exists due to virtualization, the overall database behavior remains unchanged.

It is clear from these experiments that for certain workloads and setups virtualizing off-the-shelf databases can lead to performance degradation, while for other workloads virtualization does not induce any noticeable overheads. These results motivate the need to better understand how virtualization can be efficiently used in the context of off-the-shelf databases. A more in-depth discussion of these aspects is presented in Chapter 6.

Counter arguments for virtualizing databases also point at resource overheads. The Relational Cloud [32] system makes the case for a database-as-a-service that efficiently handles multi-tenancy, has elastic scalability and supports privacy. The authors argue that achieving multi-tenancy by having a database in a VM architecture is inefficient due to additional instances of the OS and database binaries. In our own evaluations, we have not seen the memory consumption overhead of the OS and database binaries to be an issue, as it is far lower than that of the data being processed.

Given the arguments we presented in favor and against virtualizing off-the-shelf databases, we conclude that virtualization can be a suitable solution. It is a proved technology, readily available and mostly compatible with database engines. It allows fast deployments, easy manageability, implicit resource separation and offers means for online reconfiguration of the system through techniques like live migration, memory ballooning or hot plugging of CPU cores. In the context of the system presented in this thesis, we identify performance issues and functional miss-matches and provide solutions where possible.

2.5 Scaling data processing on multicores

In the previous section we motivated the need to understand performance aspects of running relational off-the-shelf databases in virtualized environments. In this section we are focusing on another problem faced by traditional off-the-shelf database engines, namely scalability on multicore machines.

Multicore architectures pose a significant challenges to existing infrastructure software such as operating systems [20, 9, 17], web servers [126], or database engines [95, 53]. The problems that multicore creates in system software – either because of the increasing number of cores [15, 3] or their potential heterogeneity [71, 58] – are by now well known.

This has triggered a lot of activity in the area of operating systems to address these problems. For instance, Boyd-Wickizer et. al [17] proposes a new exokernel
based operating system, Corey, which tries to manage the complexity of multicore machines by moving the responsibility into the application space.

Disco [21] and Cellular Disco [46], make the case for resource partitioning by running a virtualization layer over a shared memory architecture, allowing the execution of multiple commodity operating systems, and treating multicore as a cluster. Finally, [128, 9, 88] make a clear statement that multicore machines should be viewed as distributed systems and adequate algorithms and communication models should be employed.

In the case of off-the-shelf relational database engines, in spite of the intense research in the area, there are still few practical solutions that allow a more flexible deployment of databases over multicore machines. We argue that this is the result of the radical architectural changes that many current proposals imply.

It is now widely accepted that modern hardware, be it multicore, or many other developments such as flash storage or the memory-CPU gap, create problems for current database engine designs. For instance, locking has been shown to be a major deterrent for scalability with the number of cores [66]. While for a low number of hardware contexts (less than 16), locks based on Linux glibc implementation of pthreads work well, after \( \approx 40 \) hardware contexts, they spend more time blocking than spinning, leading to increased context switches. Our own experiments with MySQL presented in this section also support this argument. Another example comes from the interaction between concurrent queries when updates or whole table scans are involved can have a severe impact on overall performance [124].

As a result of the challenges posed by modern hardware architectures, a large part of the proposed work either tries to modify the database engine or to completely redesign its architecture. Just to mention a few examples, there are systems that try to replace existing engines with pure main memory scans [124]; to use dynamic programming optimizations to increase the degree of parallelism for query processing [52]; to use helper cores to efficiently pre-fetch data needed by working threads [95]; to modularize the engine into a sequence of stages, obtaining a set of self-contained modules, which improve data locality and reduce cache problems [54]; or to remove locking contention from the storage engine [66, 67].

Commercially, the first engines that represent a radical departure from the established architecture are starting to appear in niche markets. This trend can be best seen in the several database appliances that have become available [4]. For example the SAP Business Warehouse Accelerator uses a combination of column-storage, main-memory indexing, independent processing over many multicore blades, and high speed network interconnects. TwinFin from IBM/Netezza uses FPGAs between disks and CPUs to speed-up data filtering and projection, offloading query processing from the CPU to the FPGA. Oracle Exadata has a shared disk archi-
2.5. SCALING DATA PROCESSING ON MULTICORES

tecture based on Oracle RAC, and Infiniband connecting processing instances on top of the shared disk.

2.5.1 Off-the-shelf database scalability on multicores

To explore the behavior of traditional architectures in more detail, we have performed extensive benchmarks over PostgreSQL and MySQL (open source databases that we can easily instrument and where we can map bottlenecks to concrete code sequences). Our analysis complements and confirms the results of similar studies done on other database engines over a variety of multicore machines [95, 53, 67].

Load interaction

Conventional database engines assign threads to operations and optimize one query at a time. The execution plan for each query is built and optimized as if the query would run alone in the system. As a result, concurrent transactions can significantly interfere with each other. This effect is minor in single CPU machines where real concurrency among threads is limited. In multicores, the larger number of hardware contexts leads to more transactions running in parallel which in turn amplifies load interaction.

We have investigated load interaction in PostgreSQL and MySQL using the Browsing mix of the TPC-W Benchmark [121].

We ran the workload of the Browsing mix of the TPC-W Benchmark in three configurations, over a \( \approx 3GB \) dataset. PostgreSQL was running on a 48 core AMD machine. Figure 2.3 shows the throughput (in transactions per second) of PostgreSQL as we vary the number of clients issuing each of the three workload configurations: Figure 2.3(a) shows the behavior of under the full Browsing mix, Figure 2.3(b) runs the mix without the BestSellers query, while Figure 2.3(c) shows the behavior of PostgreSQL when only the BestSellers query is being executed. For all these setups, we have also varied the number of cores to which PostgreSQL was pinned.

The following observations can be made:

- For the complete mix (Figure 2.3(a)), we observe a clear performance degradation with the number of cores. We traced the problem to the BestSellers query, an analytical query that is performing scans and aggregation functions over the three biggest tables in the database. On one hand the query locks a large amount of resources and, while doing this, causes a large amount of context switches. On the other hand all the concurrent queries have to wait until the BestSellers query releases the locked resources.
CHAPTER 2. MOTIVATION AND BACKGROUND

When this query is removed from the mix, Figure 2.3(b), the throughput increases by almost five times and now it actually improves with the number of cores.

When running the BestSellers query alone (Figure 2.3(c)), we see a low throughput due to the interference among concurrently running queries and, again, low performance as the number of cores increases.

The interesting aspect of this experiment is that BestSellers is a query and, as such, is not doing any updates. The negative load interaction it causes arises from the competition for resources, which becomes worse as the larger number of cores allows us to start more queries concurrently.
Similar effects have been observed in MySQL, albeit for loads involving full table scans [124]. Full table scans require a lot of memory bandwidth and slow down any other concurrent operation, providing another example of negative load interaction that becomes worse as the number of cores increases.

### Contention

![Graphs showing database scalability](image)

(a) PostgreSQL: L2 data cache miss ratio  
(b) PostgreSQL: s_lock cache misses  
(c) MySQL: Context switches

**Figure 2.4:** Database scalability on modern multicore servers – effects of synchronization contention

One of the reasons why loads interact with each other is contention. Contention in databases is caused mainly by concurrent access to locks and synchronization primitives.
To analyze this effect in more detail, we have profiled PostgreSQL (using OProfile [91]) while running the BestSellers query. The individual run time for this query, running alone in the system, is on average less than 80ms, indicating that there are no limitations in terms of indexing and data organization. We make the following remarks:

- Figure 2.4(a) shows the L2 data cache misses for the full Browsing mix, the Browsing mix without the BestSellers and the BestSellers query alone. The L2 data cache miss ratio was computed using the expression below based on measured values for L2 cache misses, L2 cache fills and L2 requests (using CPU performance counters). We have done individual measurements for each CPU core, but as there are no significant differences between the cores, we used the averaged values of the measured metrics.

\[
L2DC\_Miss\_Ratio = \frac{100 \times L2Cache\_Misses}{(L2Cache\_Fills + L2Requests)}
\]

With more clients and cores, we see a high increase in cache misses for the workloads containing the BestSellers query. We have traced this behavior to the “s_lock” (spin lock) function, which is used in PostgreSQL to control access to the shared buffers data structures (held in shared memory). Every time a lock can not be acquired, a context switch takes place, forcing an update of the L2 cache (this depends on the lock implementation and on the hardware). The spin lock will actively spin for an amount of time, trying acquire the lock, and then will block.

- Figure 2.4(b) shows that the time spent on the “s_lock” function increases with both clients and cores, only when the BestSellers query is involved. We would expect to see an increase with the number of clients but not with more cores. Removing again the BestSellers query from the mix, we observe that it is indeed the one that causes PostgreSQL to waste CPU cycles on the “s_lock” function as the number of cores increases. Finally, looking at the “s_lock” while running only the BestSellers query we see that it dictates the behavior of the entire mix.

The conclusion from these experiments is that as the number of cores and clients increase the contention on the shared buffers significantly degrades performance: more memory leads to more data under contention, more cores just increase the contention. This problem that has also been observed and presented by Boyd-Wickizer et al. [18].
In the case of MySQL, running the same experiment, the InnoDB storage engine acts as a queuing system: it has a fixed number of threads that process client requests (these are storage engine threads). If there are more clients issuing requests than the available storage engine threads, MySQL will queue them until the previous ones have been answered. In this way MySQL is not affected by the number of clients but it shows the same pathological behavior as PostgreSQL with the number of cores: more cores result in lower throughput and higher response times.

While running this experiment we monitored the times a thread had to yield to the Operating System due to waits for a latch. Figure 2.4(c) shows the number of thread yields per transaction for different loads on the system. Based on this, we make the following observations for MySQL:

- Running a single storage engine threads for each CPU core available to MySQL, we looked at three scenarios: under-load (a total of 12 clients), perfect-load (same number of clients as storage engine threads) and over-load (200 concurrent clients). At 12 cores we see very few thread yields per transaction occurring. This indicates that for this degree of parallelism MySQL exhibits no intrinsic problems. Adding extra cores as well as placing enough load as to fully utilize the storage engine threads (i.e., perfect load and over load scenarios), we see that the number of thread yields per transaction increases significantly. We also observe that the queuing effect in the system does not add extra thread yields. With increasing cores, the contention of acquiring a mutex or a latch increases exponentially.

- Of the possible causes for the OS thread yields, we observe less than half are caused by the latches that MySQL’s InnoDB storage engine uses for row level locking. The rest are caused by mutexes that MySQL uses throughout its entire code. This implies that there is not a single locking bottleneck, but rather a problem with locking across the entire code-base, making it difficult to change the system so that it does not become worse with the number of cores.

- In the case of the BestSellers query, MySQL does not show the same performance degradation issues due to the differences in engine architectures. MySQL has scalability problems with an increasing number of hardware contexts due to the synchronization primitives and contention over shared data structures.
2.5.2 Replication within the same machine

Load interaction is an intrinsic feature of existing database engines that can only become worse with multicore. Similarly, fixing all synchronization problems in existing engines is a daunting task that requires major changes to the underlying architecture. The basic insight is that we can alleviate the problems of load interaction and contention by separating the load and using the available cores as a set of distributed resources rather than as a single parallel machine, effectively partitioning the machine.

Unlike existing work that focuses on optimizing the access time to shared data structures [53, 67] or aims at a complete redesign of the engine [54, 124], our solution does not require code modifications on the database engine. Instead, we use replicated engines each one of them running on a non-overlapping subset of the cores.

Following recent trends in operating systems that treat large multicores as distributed systems, we look at distributed database techniques (like replication) for scaling up data processing. Many database replication strategies have been developed during the last decade starting with the work on Postgres-R [68]. Of the many existing systems [26], however, not all approaches are suitable for multicore machines. For instance, many middleware database replication solutions use group communication to coordinate the copies [12, 42].

The approaches closer in design to Vela for scaling on multicores are those relying on single master replication [37, 99] and that can support specialized satellites and partial replication. This type of design is starting to be widely used in cloud computing, for instance, in the Microsoft SQL Azure database [23].

While these systems scale out their replicas over clusters, they rely on the database engine’s parallelization to scale up with the individual resources of each cluster node. As servers have increasingly more cores, many new factors impede traditional relational databases from scaling up. Among these we mention contention on synchronization primitives in a database [66, 67], workload interaction in the presence of many real hardware contexts [116] as well as the effects of hardware islands in database deployments on NUMA systems [101].

In most replication based systems, scaling out and scaling up are seen as two orthogonal problems as there is no uniform view of resources within a machine and in the cluster. We argue that using virtualization over cluster of multicores a uniform resource pool can be defined, enabling a single replication model to be used for scaling both up and out.
2.6 Dynamic reconfiguration

In the previous two sections we have presented two issues faced by relational off-the-shelf databases: performance in virtualized environments and scalability on multicore systems. In this section we introduce a third issue, that of dynamic reconfiguration of the resource allocation of relational off-the-shelf databases, in the context of virtualization.

Dynamic system reconfiguration refers to the system’s ability to change at runtime at different levels of granularity. We differentiate between coarse grained reconfiguration operations (like creation or removal of VMs or VM live migration) and fine grained reconfiguration options (like hot plugging and removing of memory, CPUs, network interfaces or block devices).

2.6.1 Coarse grained reconfiguration

Solutions like SCADS [123] or that of Lim et al. [73] that do automatic scaling only do coarse grained operations by adding / removing physical servers in the system. DeepDive [89] reconfigures the collocation of VMs in case of performance interference – a problem known as “noisy neighbor”. A noisy neighbor a service that through its resource usage pattern puts large demands on the server and negatively impacts the performance of collocated services.

A series of virtualized database live migration solutions also address the noisy neighbor problem. SWAT [83] resolves the problem of performance interference among tenants through a load balancing method based on database replica swap. In their replica swap approach they have a single master asynchronous replication scheme (similar to Vela), but use the replicas only for availability – meaning that contrary to Vela, their replicas do not process any of the incoming load. When the master becomes overloaded, the roles of the master and secondary replicas are swapped, relieving the master’s resource pressure by moving it to a secondary replica.

Albatross [36] is a technique developed for enabling live migration of database instances processing OLTP workloads using network attached storage as a medium for persistent dataset storage. They avoid traditional full VM migration arguing the unnecessary cost of migrating OS and database binaries. Their approach to migrating the database caches, active state and transactions is based on creating a snapshot of a database, draining the current transactions and their changes to the new snapshot and doing an atomic handover from the old database to the new snapshot. Such an approach might be an interesting option for live-migrating in Vela.
Other approaches to the noisy-neighbor problem use the concept of resource slack [8] for determining when a virtualized database should be migrated, either through cold or hot migration mechanisms. Through defining the available “slack” and load throttling, the negative performance impact of the migration process can be minimized.

ProRea [106] is a technique for live-migration of virtualized relational databases engines relying on snapshot-isolation. Resembling Albatross, ProRea does the database migration in a series of steps. The “hot page push” is when the buffer pool pages are transferred to the destination which takes ownership of the pages that are not involved in transactions at the source. New transactions are processed at the destination and the existing ones are drained at the source. This phase is the “parallel page access” in which a consistent state among the pages is ensured. The final phase of “cold pull” occurs once all transactions on the source finish. During this phase the destination transfers all remaining pages from the source. The ProRea might be well suited for live database migration, but requires a very specific database implementation. Also, their evaluation is only on small buffer pools (1.5GB). The impact of increased buffer pool sizes and high loads diminishes the gains of such an approach compared to database agnostic virtualization based live migration.

2.6.2 Fine grained reconfiguration

All the previous solutions do not cover fine grained reconfigurations like hot plugging or removing of memory or cores in VMs. Other approaches study finer grained online reconfiguration. For instance Microsoft SQL Server’s dynamic memory configured on top of Hyper-V [104] and Application Level Ballooning as a generic memory ballooning mechanism implemented for MySQL [115] demonstrate that memory can be dynamically added or removed from databases running in VMs. Neither however takes into account coarse grained operations.

Of interest in the context of this thesis is fine grained memory reconfiguration in virtualized environments, targeting system software like databases and language runtimes that do their own memory management.

Virtualization in cloud computing and server consolidation enables applications previously deployed on dedicated machines to share a physical server, reducing resource consumption, energy, and space costs among other benefits. Statistically multiplexing such servers and ensuring application performance as load changes, however, requires careful coordination policies, and virtual machine monitor (VMM) mechanisms like live migration and memory ballooning are used to
These techniques work well when the OS manages application memory via paging. However, they do not work well when the application manages memory itself—the common case in server applications like databases and language runtimes. Here, efficient execution depends on the program having an accurate picture of available resident memory. Dynamically reconfiguring these systems, by reallocating RAM using standard ballooning severely impacts performance, leading to thrashing and, in some cases, failure. This has also been reported in recent publications as a limiting factor in the context of database consolidation in virtualized environments. While Curino et al. [33] report possible database consolidation rates of up to 17:1 for certain workloads, they note that in general existing memory reconfiguration techniques (like OS ballooning) are useless as they are not integrated with the database’s buffer pools.

A database, for example, allocates memory internally from a large, fixed size pool acquired at startup which it assumes is mostly resident. Standard ballooning transparently pages this pool to disk, causing the DB to choose the wrong query plans as data it thinks in main memory is actually not there. Figure 2.5 shows this effect when running a query from the TPC-H Benchmark [119] on MySQL. The figure shows the response time for the query with the database operating on 4, 6,
and 8 GB of actual machine memory and when the memory has been ballooned down from 10 GB to 4, 6, and 8 GB using standard ballooning techniques [7]. That is, in each pair of experiments, the database has the same amount of machine memory. However, in the case of ballooning, the fact that some of the memory it thinks it has is actually not there has a catastrophic effect in performance. This effect is well known in industry and has already been mentioned by other researchers [33].

We focus on applications that manage their own memory, are statically configured with a fixed quantity of memory, and optimize their use of this pool to maximize performance. Most enterprise applications fall into this category. Here we consider databases, which extensively cache data and results to avoid I/O, and language runtimes, which exploit extra memory to reduce the frequency of expensive garbage collections.

Ballooning is useful if varying the actual memory allocated to an application changes its performance in a stable way. That such is the case for sever applications like databases and language run times is easy to demonstrate. Figure 2.6 shows the response time for the XMark XML benchmark [107] as we vary the configured JVM heap size: additional memory dramatically reduces the overhead of garbage collection, resulting in significant overall speedup although the amount of actual effective work done remains the same.

Figure 2.7 shows how the memory size of a database running the TPC-H bench-
mark \[119\] affects query response times (note the log scale). Increasing available memory from 4GB to 8GB reduces the response time of most (but not all) TPC-H queries, sometimes by two orders of magnitude, mostly due to reduced I/O. Databases manage their own pool of memory because they have knowledge of access patterns and relative page utility, knowledge not available to the OS.

Such a significant influence leads to two problems with statically provisioning RAM to VMs, even when informed by load profiles \[110\]. The first is the need to provision for peak load, leading to inefficiencies as it is expensive to leave memory idle, the space-time tradeoff depends heavily on workload, and less applications can be collocated in a physical machine. The second is fragmentation when starting, stopping, or migrating VMs, which creates a packing problem across physical machines.

Different solutions have been proposed for databases and runtime systems to overcome this problems.

MEB \[133\] does dynamic memory balancing for VMs based on runtime statistics from the VM kernel in order to optimize performance of memory dependent applications. MEB uses OS-level ballooning in Xen to re-configure memory. Figure 2.5 shows that this is not applicable to applications which manage their own memory, though it might fare better with some systems like PostgreSQL which use the OS disk cache for this purpose.

![Figure 2.7: Effect of DBMS memory size on TPC-H query response time](image-url)
CHAPTER 2. MOTIVATION AND BACKGROUND

Hines et al. [61] present policy framework (Ginkgo) which correlates application performance to its memory needs in order to satisfy capacity and performance constraints using runtime monitoring. They dynamically resize the JVM heap using a balloon process that allocates and deallocates memory through JNI. Also, a recent patent from VMware [78] describes ballooning in a Java VM by allocating Java objects. Both Ginkgo and the VMWare patent approach the problem by being garbage-collector-agnostic. We consider that such approaches suffer from a series of limitations. First, translating from an object reference to an OS virtual address, without modifying the JVM, requires calling into native code via JNI (since the object address is not available within Java). Second, most GCs move objects among and within heap spaces, requiring re-translation of the object address after each collection phase. Although Java VMs (like OpenJDK) offer runtime callbacks for garbage collection completion, such an approach would still result in considerable runtime overhead in remapping pages. Alternatively, objects used for memory reservation could be pinned in memory using JNI, though the implications of this vary between GC implementations. Consequently, it is not possible to be fully independent of the garbage collector.

Microsoft’s SQL Hyper-V/VM [104] permits dynamic memory configurations for SQL Server, but does not present any details on how this is achieved and how the database reacts to memory changes. In [86], SQL-VM is presented as a solution for resource (CPU, I/O and memory) sharing in multi-tenant database-as-a-service systems.

In contrast to white-papers and patents form the industry, we present a detailed technical description and reproducible performance analysis of a method for achieving dynamic memory reconfiguration in virtualized environments for systems that do their own memory management. We refer to this method as Application Level Ballooning and we describe it in detail in Chapter 5. We show that our approach is generic and can be extended to any applications managing their own memory (whether a buffer pool or runtime heap), whereas VMware focuses only on a JVM heap and HyperV only on the MS SQL database.

Orthogonal to the above approaches, modern VMMs also employ other techniques. For example, content-based memory de-duplication reduces memory consumption by copy-on-write sharing of identical pages across the VMs [81, 127, 51]. Heo et al. [56] use control theory to meet memory utilization targets when overbooking memory with ballooning in Xen-based consolidation environments. The approach requires measurable memory demands to drive the control loop. Autocontrol [94] also applies control theory to optimize resource assignment to VMs, focusing on CPU and disk bandwidth.

\[1\] JNI [64] is the Java Native Interface – a framework that enables Java applications to have components natively built for specific platforms.
2.7. **CHAPTER SUMMARY**

2.7 Chapter summary

In this chapter we discussed how the “one-size-fits-all” philosophy no longer applies to databases. Database systems are diverging as a result of increased workload variety, vast hardware diversity, and diversified business requirements. This divergence is leading to specialized solutions that address specific workloads under well defined business requirements. With this in mind, the solution presented in this thesis in the form of the Vela system aims at addressing a wide set of transactional read-intensive workloads, on modern multicore hardware.

Current cloud-enabled data processing solutions tend to have restricted APIs, reduced consistency support or limited functionality. These limitations impose extra engineering effort in porting existing applications to the cloud. Through Vela we provide a solution that does not limit the querying API (providing full SQL support), offers Snapshot Isolation consistency and supports the full feature set of the off-the-shelf databases it relies on.

Vela aligns itself to the characteristics of scalability and flexibility, common for cloud solutions. Using a replication model (i.e., single-master replication), Vela scales data processing out over clusters of dozens of nodes, but at the same time scales up on large multicores. To the best of our knowledge it is the first system that reacts to the number of nodes and their size (in terms of cores and memory) using the same replication model. While standalone off-the-shelf databases have difficulties in harnessing large multicore servers due to load interaction or synchronization contention, the replication model of Vela solves both problems.

Most commercial cloud infrastructures (e.g., Amazon EC2, Rackspace, Windows Azure) rely on virtualization. As such, we treat virtualization as a first class citizen in Vela. Through virtualization our system gains a uniform view of the resources across different servers and within large multicore servers. Naïvely deploying off-the-shelf database engines in virtualized setups may often lead to degraded performance and functional miss-matches. In designing and building our solution we take these into account, in order to be able to overcome them. One functional aspect that we focus on in the context of Vela is the support for dynamic memory reconfiguration. For efficiently consolidating multiple database instances of Vela within the same multicore server, existing mechanism are not suitable. As a solution we champion Application Level Ballooning, a dynamic memory reconfiguration mechanism for virtualized applications that manage their own memory (e.g., databases and language runtimes).
Vela is a system designed for scaling transactional data processing of read-intensive workloads on multicores as well as clusters of multicores. Vela relies on data replication as a model for scalability and on virtualization as an enabling technology. Both the replication model and the use of virtualization are further detailed in this chapter.

Vela extends Multimed [116], a system that I have previously built. Multimed was designed for scaling transactional data processing on large multicores. Vela adds support for distribution over clusters of multicores, automatic and online reconfiguration, support for virtualization and multitenancy.

Multimed was inspired by work in multikernel operating systems [76, 88, 128, 9] which consider representing a multicore machine as a distributed system. In Multimed, a database is deployed on a multicore machine as a collection of distributed replicas coordinated through a middleware layer that manages consistency, load balancing, and transaction routing. In other words, rather than redesigning the database engine, we partition the multicore machine and allocate an unmodified database engine to each partition, treating the whole as a distributed database.

As Vela incorporates Multimed, its feature set being a super-set of those in Multimed, this chapter describes only Vela.
3.1 Design goals

Vela, a transactional data processing system, has the following design goals aimed at enabling it to be a cloud-ready solution based on off-the-shelf database engines.

**Scalability:** The system scales both with increasing load and with increasing hardware resources. Scaling with load requires the system to react to the increasing number of clients as well as to different concurrent workloads. Scaling with hardware resources requires Vela to scale both *up* on large multicores and *out* on clusters.

**Flexibility:** The system permits online reconfiguration in order to react to changes in load or available resources. Vela offers a reconfiguration API that can be used manually or by a policy framework. Policies running on top of the system enable automatic system reconfiguration.

**Multitenancy:** One instance of the system, spanning over a cluster of multicores can host multiple tenants, each having its own database. The resources available to Vela can be re-allocated among tenants, while otherwise the tenants run isolated.

3.2 Data replication

Vela scales both within the boundaries of the same multicore machine and in a cluster of multicores by relying on primary-master data replication, similar to the Intra-Stamp replication of WAS [22], Ganymed [99] or the fork/follower replication of Heroku Postgres [57]. The table in Figure 3.1 summarizes some of the similarities and differences between these replication solutions and Vela, while the rest of this section talks in more details about these solutions.

<table>
<thead>
<tr>
<th>System</th>
<th>Replication for perf.</th>
<th>Replication for avail.</th>
<th>Scale out</th>
<th>Scale up</th>
<th>Online reconfig.</th>
<th>Virtualization ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Azure</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>limited</td>
<td>✓</td>
</tr>
<tr>
<td>Ganymed</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Heroku Postgres</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>limited</td>
<td>✓</td>
</tr>
<tr>
<td>Vela</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Figure 3.1:* Overview of replication systems feature-sets

SQL Azure [22] does not emphasize virtualization or flexibility in the replication configuration – having fixed number and scope of replicas. Its Intra-Stamp replication offers fault tolerance and load balancing by having a primary master and
two secondary replicas, synchronously replicated. The Inter-Stamp asynchronous replication is used for geo-replication and disaster recovery, as well as a mechanism for data migration. The authors of [59] investigate the performance of SQL Azure as a black box. The points of interest are the network interconnect inside the cluster (peaking 95% of the time at 90MB/sec) and the cost of a TCP RTT, which is below 30 milliseconds for 90% of the samples. Also, under a TPC-E [118] like workload, they observe that the system scales up to 64 clients. In our evaluation we show that Vela can scale beyond this, in an environment with a similar network configuration.

The Ganymed system [99] has a similar replication model to Vela, but only emphasized the benefits of scaling-out over clusters by load-balancing the read-workload over replicas. The system is static in nature, not providing any form of reconfiguration support. Also, the system is tightly coupled in its design to PostgreSQL, making it a poor candidate for supporting various off-the-shelf databases. In contrast to Ganymed, Vela allows data processing to be scaled both over cluster and within individual multicore servers while supporting different database engines and providing a rich reconfiguration API and support for virtualization.

The Heroku Postgres solution [57] closely resembles Ganymed, by relying on unmodified PostgreSQL database engines and in the employed replication model. Both Ganymed and Heroku Postgres rely on primary-master lazy replication for load balancing read intensive workloads over clusters. As an industry solution, Heroku Postgres provides a functional solution based on Amazon’s EC2 virtualized infrastructure. Fine grained system reconfiguration, automatic scaling based on load changes, scalability on large multcore machine, support for different database engines or multitenancy are features that are missing in Heroku Postgres.

In primary-master replication, there is a primary copy of the data and a series of replicas of the primary copy. In this thesis we use the terminology of “master” to refer to the primary copy and “satellite(s)” to refer to the replicas. All update transactions go to the master while read transactions go to the satellites which are kept in sync with the master. Upon committing an update transaction the master’s server count number (SCN) is incremented and its change set is extracted and propagated to all satellites along with the SCN. Satellites increment their SCN after applying the changes from the master in the same order. Each transaction is tagged with the current SCN of the master when entering the system.

Vela guarantees snapshot isolation as its consistency level. When using snapshot isolation the queries are guaranteed to see all changes that have been committed at the time the transaction they belong to started (this is a form of multiversion concurrency control that is present in many off-the-shelf database engines such as Oracle, SQLServer, or PostgreSQL).
Moreover, within each replica, Vela relies on the snapshot isolation consistency of the underlying database engine. This has the advantage that update transactions do not interfere with read transactions. The update transaction is applied on a different snapshot of the database and once it is committed, the shadow version of the data is applied on the active one. In this way, Vela can schedule queries on replicas at the same time they are being updated.

This primary-master replication model is suitable for a large set of read-intensive workloads, though less applicable for update-mostly scenarios.

Vela uses lazy replication [47] between its master and satellites but guarantees a consistent view to the clients. Figure 3.2 gives an overview of how replication works in Vela. The master is responsible for keeping a durable copy of the database which is also guaranteed to hold the latest version of the data. All the update transactions are executed at the master as well as any operation requiring special features such as stored procedures, triggers, or user defined functions.

The satellites hold replicas of the database. These replicas might not be completely up to date at a given moment, but they are continuously fed with all the changes done at the master. A satellite may hold a full or a partial replica of the database. Doing full replication has the advantage of not requiring knowledge of the data allocation for query routing. On the downside, full replication can incur
higher costs in keeping the satellites up to date due to larger update volumes, and lower performance because of memory contention across the replicas. In the experimental section we include an evaluation of partial replication but all the discussions on the architecture of Vela are done on the basis of full replication to simplify the explanation.

When a query enters the system, the Router needs to decide where it can run the transaction (i.e., to which replica the incoming request should be bound to). The binding process may involve some small delay until a copy has all necessary updates. If multiple options are available, the Router chooses the one with the smallest load. Note that the master is always capable of processing any request without any delay and can be used as a way to minimize latency if that is an issue for particular queries. Details on how the Router chooses between the viable replicas and the master are further detailed in Section 3.3.1.

Each time an update transaction is committed, the master commits the transaction locally. The Router then extracts the changes and propagates them as a list of rows that have been modified (black dashed lines in Figure 3.2). A satellite enqueues these update messages and applies them in the same order as they were executed by the master.

In order to capture the changes caused by an update transaction, Vela uses row-level insert, delete and update triggers in the master database on the union of the tables replicated in all the satellites. The triggers fire every time a row is modified and the old and new versions are stored in the context of the transaction. All the changed rows, with their previous and current versions, represent the WriteSet of a transaction. In our system, this mechanism is implemented using SQL triggers and server side functions. This is the only mechanism specific to the underlying database but it is a standard feature in today’s off-the-shelf database engines.

Figure 3.3 shows the algorithm that handle transaction commits. It describes the transaction commit operation performed at the master along with the WriteSet extraction and propagation. When an update transaction is committed on the master, the WriteSet is extracted (by invoking a server side function) – line 1. If there are changed tuples then we acquire a global lock on the master (line 3), ensuring that the order in which we commit the transaction at the master is the same order in which we apply the WriteSets on all satellites. Within the scope of the lock (lines 4-8), we increment the Server Count Number of the master and tag the WriteSet with the new value. The WriteSet is then propagated to all satellites by placing it on their processing queue (lines 7-8). Locally, each satellite dequeues WriteSets and atomically applies it to its underlying database. Through this process, the satellites are asynchronously receiving all the updates from the master but are guaranteed to see them in the same order. The asynchronous
replication allows the master commits to complete without waiting for the satellites to also commit the changes – keeping the update transaction latency low. If the WriteSet is empty (line 10) then the update has not modified any tuples in the database and there is nothing to propagate to satellites. The transaction only needs to be committed at the master.

A total order is enforced by Vela over the transaction commits at the master. This order needs to be enforced so that it can be respected on the satellite as well. This might be a performance bottleneck for the system, but as we show in the experimental section, the overhead induced by WriteSet extraction and by enforcing a total order over the commits of updates is small. In practice, Vela introduces a small latency in starting a query (while waiting for a suitable replica) but it can execute many more queries in parallel and, often, the execution of each query is faster once started. Thus, the result is a net gain in performance.

Based on the Snapshot Isolation offered by each database and on the mechanism of extracting, tagging and propagating the updates from the master to satellite replicas, each client is guaranteed to see the changes that it previously did within the current transaction.

3.3 System components

At a high level Vela is composed of different Data Processing Instances (DPIs) and a Router. The complete view of the system’s state that maps the DPIs and the Router to physical resources is stored in the System Model.
3.3. SYSTEM COMPONENTS

3.3.1 The router

The Router faces the clients accepting requests and returning responses. Requests are queued and then handled by worker threads. Worker threads dispatch requests to the database instances in the DPIs for the actual processing. The Router is also in charge of extracting the changes in the data on the Master (i.e., the WriteSets) and propagating them to the other DPIs in the system.

```java
LOADBALANCING(Transaction tx, SystemModel sm, tries, delay)
1  dpi ← null
2  for i in tries
3     do
4       minLoad ← ∞
5       crtLoad ← 0
6       for sat in sm.GETSATS(master)
7          do
8             if sat.SCN >= tx.SCN
9                 then
10                if sat.SUPPORTS(tx) or !tx.NEEDSPARTIAL()
11                   then
12                      crtLoad ← sat.GETLOAD()
13                      if crtLoad < minLoad
14                        then
15                            dpi ← sat
16                            minLoad ← crtLoad
17                if dpi! = null
18                  then RETURN(dpi)
19                  SLEEP(delay)
20             RETURN(master)
```

Figure 3.4: Router load balancing algorithm

The Router also balances load among replicas of the same tenant by choosing the least loaded satellite of those that are able to handle the transaction. The algorithm shown in 3.4 exemplifies this. The ability of a satellite to handle a transaction is given by its SCN and by the actual data it holds (in the case of partial replication). Line 6 iterates over all the satellites of a master. The check on line 8 ensures that the satellite has the required updates for processing the transaction (it has applied all the WriteSets up to the SCN with which the transaction was tagged when it entered the system). The partial replication requirements are checked on line 10. If the currently checked satellite is a partial replica we check if it supports
the current transaction. If it is either a full replica or a partial replica that supports
the current transaction, we compare its current load to the minimal load found so
far (lines 12–16). If a suitable satellite was found, the we route the transaction “tx”
to the Data Processing Instance of this satellites (line 18). Conversely, we repeat
the process for a total of “tries”-times with a “delay” between each repetition. If
after \(tries \times delay\) units of time we were unable to discover a satellite to handle
the transaction, we assign the transaction to the master (line 20). Tuning “tries”
and “delay” values is an interesting aspect as they directly determine the added
latency to a read-only transaction. The “delay” is highly correlated with the
WriteSet extraction and propagation time. We found this to be less than 10
milliseconds, even on a highly contended master. Heuristically, we discovered that
the “tries” count should keep the maximum total added delay below half of the
average read-only transaction latency: \(tries \times delay < \frac{\text{latency}_{avg}}{2}\).

The load metric reported by the satellites (the “getLoad” call on line 12) can be
any number of system or environment metrics: the number of active transactions,
the CPU usage, average run time on this DPI, etc. For all the workloads that we
studied, we found the simple heuristic of the active number of transactions for a
satellite to work best. We do not rely on the actual number of active transactions
as that would require an atomic counter. An approximation based on connection
pool statistics from each DPI proved sufficient.

Read-only transactions are discriminated in the Router based on the transac-
tion’s readOnly property from the JDBC API.

The Router component is implemented in Java 7 around the Apache Mina [5]
library for network I/O. A configurable number of Mina NIO threads handle client
I/O through polling. The number of NIO threads (equal to the number of cores
allocated to the Router) queue incoming requests. These are processed by worker
threads that dispatch the requests to the database instances through native JDBC
connections.

A control thread is spawned in the Router for each DPI. For the master, the
thread extracts the WriteSets and propagates them to each DPI. In the case of
the replicas, the control thread applies WriteSets and keeps track of the replica’s
version (i.e., SCN).

The messages received by the Router from the Clients (through a custom JDBC
Type 3 Driver) are on average under 100 bytes. By default, multiple messages will
be packed together before being sent (based on Nagle’s algorithm [98]), increasing
the response time of a request. We disabled this throughput optimization favoring
latency, by setting the TCP_NODELAY option on the Java sockets. This reduced
the RTT for messages by a factor of 10 at the cost of a higher number of packets
on the network. The impact of this in the case of virtualized DPIs is discussed in detail in Chapter 6.

All connections from the DPIs to their underlying database engines are done through JDBC Type 4 Drivers (using native protocols) to ensure the best performance. Using our own connection pool increases performance as no wait times are incurred by expensive operations of creating/freeing a database connection.

3.3.2 The data processing instances

Each DPI (Data Processing Instance) is backed up by a database instance running on top of a copy of the data and is allocated a set of resources (CPU, memory) which can be changed in runtime. Like Amazon RDS or SQL Azure, Vela uses existing relational database engines in its design. In the case of Vela we chose PostgreSQL and MySQL as supporting databases. Replacing them with other database engines is straightforward as the only engine specific part of Vela is the WriteSet extraction mechanism.

The DPIs in Vela can have different types: Master, Satellite, HotBackup or Observer. Vela supports multi-tenancy by allowing multiple Masters. Each Master has its own set of Satellites, HotBackups and Observers. Each database instance underlying in a DPI has two data components: the actual data and the transaction logs. The options for storing the two can be specified for each replica as one of the following: main memory (RAM), local disk or remote network attached storage (NAS).

Master DPIs

The Masters are primary copies of the data and are configured to offer durability. The Router dispatches each tenant’s update transactions to its Master. The Masters are configured as durable, with the transaction log on a local disk and the data on either local disk or NAS. Alternatively, in a non-durable setup the Master can be entirely held in RAM. The non-durable setup is not recommended and only used for evaluating Vela’s scalability without having the transaction log as a bottleneck. By storing the data and the transaction logs separately we have observed higher performance gains and more stable behavior of the Master (as mentioned in other systems, e.g., the journaling drive in SQL Azure [22]).

Satellite DPIs

The Satellites hold replicas of the data of each Master. The Router assigns the read-only transactions to Satellites that have the SCN matching the transaction’s
SCN. When more Satellites fulfill this condition, the one with least load is chosen. Satellites do not need to ensure durability (but could be configured to do so).

The Satellites can have both their data and transaction logs in RAM. As most database engines do not have an option for disabling the transaction log, we emulate this by storing it in RAM and disabling synchronous commits. The overall system durability is given by the Master. For instance, for the PostgreSQL satellites, we turned off the synchronous commit of transactions and increased the time until these reach the disk. Consequently, the PostgreSQL specific options like \textit{fsync}, \textit{full_page_writes} and \textit{synchronous_commit} were set to \textit{off}, the \textit{commit_delay} was set to its maximum limit of 100 milliseconds, and the \textit{wal_writer_delay} was set to 10 seconds.

While the transaction logs are small enough to be always held in RAM, the actual data might be too large for this. In this case, the data is held on NAS and the available RAM is used for caching. The reconfiguration API supports adjusting the memory of each DPI for increasing or decreasing the memory cache. As PostgreSQL relies on the disk cache for most of its data caching it requires no modifications to be able to adapt to changes in the DPI’s memory. For database engines that do their own memory management, like MySQL, a virtualization based solution like Application Level Ballooning (ALB) \cite{115} can be used.

From the data storage and durability perspective, we identify three configuration optimizations levels:

- Configuration optimization level \textbf{C0 (No Optimization)} implements full data replication on durable local storage for all Satellites. This is approach is suitable for low memory scenarios in which the Satellite DPIs can not be allocated sufficient RAM to hold the data in main memory. The bottleneck for such an approach is persistent storage (e.g., local disk) for which multiple database engine will contend. Given separate disks for each DPI, the contention can be minimized.

- Configuration optimization level \textbf{C1 (Main Memory)} implements full data replication in main memory for all Satellites. With this optimization high-performance disks are traded for main-memory, sufficient to store multiple DPIs completely in RAM. With the C1 optimization, all disk contention is removed. It is not always necessary to hold the entire dataset in main-memory. A compromise between memory requirements and performance can be achieved for most workloads (e.g., caching 80% of the data instead of 100% would only lead to a small performance overhead). Using the dynamic reconfiguration API described in Section 3.5, memory changes at runtime (based on Xen’s memory ballooning or Application Level Ballooning) can offer the whole optimization spectrum between C0 and C1.
3.3. SYSTEM COMPONENTS

- Configuration optimization level **C2 (Partial Replication)** implements partial data replication in main memory for the Satellites and transaction routing at the **Router**. This approach uses far less memory than C1, but requires a-priori knowledge of the workload to partition the data adequately (Satellites will be specialized for running only given queries). Extensive research in the area focuses on automatic partitioning that does not require client a-priori knowledge. Ideas from systems like SWORD [102] or Schism [34] could be plugged into Vela’s existing support for partial replication.

In evaluating Vela, we analyzed all the three configuration optimizations (C0, C1 and C2) for scaling up the system as well as discrete points in the spectrum C0 – C1.

**Observer DPIs**

The Observer is used for determining the best response time for the read-only workload. The Observer is an optional DPI, always deployed on 1 core (exclusively assigned to the Observer DPI). The Router ensures that if an Observer DPI is present, it will always have one and only one transaction routed to it (this is done by the Router’s load balancing algorithm). As only one transaction is being executed at a time there is no contention in the Observer DPI. Since the Observer DPI is sampling response times and does not process concurrent requests, it has no scalability requirements and can be configured to run on only 1 core. The Observer approximates the best transaction latency for the current read-only part of the workload and reports it as a metric that can be used in target functions.

**HotBackup DPIs**

The HotBackups are special and optional type of DPIs. They do not process client requests, though they are kept up to date, receiving all the WriteSets. HotBackups are used for spawning new Satellites at runtime for each tenant.

The HotBackup does not handle incoming workload. It is used as a checkpointing mechanism for the Master. The HotBackup receives all the WriteSets generated by the Master. When a new Satellite is added to the system, the following steps are taken.

1. The Router freezes the HotBackup by not sending it any more WriteSets and shuts down its database engine.
2. The Router starts accumulating the WriteSets for the HotBackup in a main-memory queue (if the Router fails, no data is lost as the Master offers durability).

3. The HotBackup data and transaction logs are copied into a new Satellite. Upon completing the copy operation, the new Satellite and the HotBackup are started.

4. The main-memory queue of WriteSets is drained into both the new Satellite and the HotBackup, allowing them to catch up with the Master.

5. Once the DPIs catch up with the Master, the new Satellite starts receiving load and the HotBackup starts checkpointing the Master again.

The HotBackups enable adding new Satellites without taking the whole system offline. The overhead is minimal in terms of resources: the HotBackup requires only 1 core and main memory for the transaction log. The time it takes to copy the HotBackup into a new Satellite is bound by the network bandwidth. This time only influences the amount of main-memory needed by the queue that temporarily stores pending WriteSets for the HotBackup and new Satellite. The maximum size of the queue can be approximated:

\[
\frac{Tx}{sec} \times \frac{avg(WriteSetSize)}{Tx} \times CopyTime(sec)
\]

For a tenant processing 10'000 transactions per second, with an average of 1KB of WriteSet data per transaction, deploying a 20GB dataset over a 1GBit network would grow the queue to a maximum of \( \approx 1.5GB \) – which is easily manageable.

In our evaluation of the system, we show that for common workloads, the time it takes to add a new Satellite to the system (copying, starting and catching up with the Master) is less than 5 minutes, being dominated by the data copy operating.

Alternatively, a Satellite can be used instead of the HotBackup. We consider this design to be less favorable: for reacting to increasing load we would first have to induce a performance drop (by taking a Satellite offline) before we can add a new one.

### 3.3.3 The system model

The System Model describes the mapping of hardware resources of all DPIs, as well as the mapping DPIs to each Master. It is used by the Router when processing client requests.
3.3. SYSTEM COMPONENTS

The System Model is represented as an XML document that is fed to Vela on start-up and is manipulated in run-time as a consequence of online reconfiguration. It describes the physical location of data and transaction logs for the DPIs, along with software and hardware resources.

Figure 3.5 depicts a snippet of a System Model XML document. The System Model XML snippet is as follows:

```xml
<system>
  <DPIs>
    <DPI><name>DPI-Master-TPCW</name>
      <physical>
        <cpus>
          <cpu><frequency>2400</frequency></cpu>
        </cpus>
        <tasks></tasks>
        <memorySize>1024</memorySize>
        <caches>
          <cache><size>16</size><level>1</level></cache>
          <cache><size>2048</size><level>2</level></cache>
          <cache><size>6144</size><level>3</level></cache>
        </caches>
      </physical>
      <logical>
        <replicationType>2</replicationType>
        <dbType>1</dbType>
        <supportedTransactions>-1</supportedTransactions>
        <poolSize>200</poolSize>
        <jdbcDriver>systems.jdbc2v1.Driver</jdbcDriver>
        <userName>username</(userName>
        <password></password>
        <database>tpcw</database>
        <mode>2</mode>
        <hostName>10.111.1.152</hostName>
        <hostPort>5432</hostPort>
        <location>/mnt/NAS/db</location>
      </logical>
    </DPI>
  </DPIs>
</system>
```

Figure 3.5: System Model XML snippet
Model comprises a collection of “DPIs”, where each “DPI” entry represents one Data Processing Instance of Vela. A DPI is characterized by a physical and a logical descriptor. The physical descriptor contains information about the cores, memory and caches that are supporting the DPI. These information can be used for example by the Router in the load-balancing requests over multiple Satellites. The logical descriptor holds information about the underlying database engine type (“dbType”), the replication optimization used (“replicationType”), data files location (“location”) as well as connectivity information (“userName”, “password”, “database”, etc).

### 3.4 Virtualization

In Vela, all the Data Processing Instances of the system are spread out over the available resources in a cluster of multicores. We have looked at different options for managing the resources (CPU, RAM) of a cluster of multicores and formulate Vela’s requirements for a resource management system:

- **Multiplexing** the hardware for running multiple Data Processing Instances on the same physical machine while allowing for NUMA-awareness. This enables Vela to scale up data processing.

- **Isolation** of CPU and memory allocated to each Data Processing Instance, such that it can make assumptions on the current available cores and memory.

- **Reconfiguration** at runtime of the resources allocated to Data Processing Instances.

- **Uniform view** of the whole cluster. Migrating a Data Processing Instance from one cluster node to another should be seamless and with no service interruption.

Two options prevail: virtualization or a custom resource management systems for large clusters.

Custom resource management and task scheduling in large datacenters is the focus of projects like Google Omega [108] or Apache Mesos [60]. While these approaches are generic frameworks for running varied jobs on datacenters they are not designed for relational data processing and might require re-engineering the database engines. Both projects avoid virtualization for the purpose of managing the global set of resources in a data center, arguing that seemingly small resource overheads can be huge at scale.
Virtualization is a good choice being a proved technology, readily available that requires no modifications to the existing database engines. It allows fast deployments, easy manageability, implicit resource separation and offers means for online reconfiguration of the system through techniques like live migration, memory ballooning or hot plugging of CPU cores. Two issues arise from using virtualization.

First, is a possible waste of resources due to the virtual machine monitor and running multiple OSes on the same machine. Second, is the need of a middleware component that coordinates the whole system. Regarding the first issue, we have not seen problems at the scale at which we ran Vela. The second issue is overcome by Vela’s design, as the Router has all the required knowledge for managing the resources of all DPIs.

In Vela we opted for virtualization (Xen Hypervisor [7, 130]), as it fulfills our requirements. Nonetheless, the resource management mechanism is orthogonal to the replication model of Vela: replacing virtualization in favor of another resource management solution is an interesting avenue to pursue in the future.

The guest domains (DomUs) run in paravirtualized mode. We have observed no impact on the performance of the Satellites that have the dataset in main-memory. For the Master and the Router that are I/O intensive, the DomUs in their default configuration have quickly become the bottleneck. Similar observations [80] indicate that network I/O can easily become a bottleneck.

Latency wise, we were able to tune the system by Linux Kernel network specific settings in both Dom0 and DomU. Dom0 sets the Transaction queue length for the NIC (txqueuelen) to 1000, uses the reno TCP congestion control mechanism and enables Generic Receive Offload (gro). For DomU, we saw benefits from increasing the maximum network read and write memory to 128 MiB (net.core.rmem_max, net.core.wmem_max), using the cubic TCP congestion control mechanism and enabling the Generic Segmentation Offload (gso). These settings follow some of the Xen networking performance guidelines [131].

CPU wise, the problem was more involved. For network I/O, Xen allocates in the host domain (Dom0) one kernel thread for each virtual network interface (VIF) in the DomUs. DomUs hosting Routers that do high frequency network I/O were provisioned with up to 8 VIFs. In order to remove the CPU bottleneck of the VIF backing threads in Dom0 we allocated a number of cores equal to the total number of VIFs of all DomUs. This way the CPU cost is balanced over all available cores. Xen allows dynamic changes in the number of cores in each domain (including Dom0). Using udev rules we hotplug and remove cores at runtime.

The multiple VIFs of the same link were bonded using Linux’s network bonding driver with a round-robin policy.
The interrupts generated in Dom0 and DomUs need to be balanced over all available cores. Failing to do this will lead to one core handling most IRQ requests, becoming a bottleneck. Manual IRQ balancing is possible, but we have found that an approach based on the IRQ Balancing Daemon `irqbalanced` can monitor and balance the interrupts automatically, even as cores are added or removed from the system.

The issues described here are in very Xen-specific terms for clarity, but corresponding concerns apply in equal measure to other hypervisors.

While Vela can run completely unvirtualized, spanning from a single machine to a cluster of machines, virtualization adds much more flexibility in its design, as we discuss in the next section.

In building Vela, we found that the benefits of virtualization (e.g., isolation, resource partitioning, support for online reassignment of memory and cores) for the automatic reconfiguration are not for free. Intrigued by the possible performance overheads we did an in-depth analysis of the impact of virtualization on the performance of database engine, from the perspective of Vela’s design, presented in Chapter 6. At a high level we remark that there is no performance penalty for the Satellites (CPU and memory bound) and that the performance of the Masters and the Router (disk and network I/O bound) might affected.

### 3.5 Dynamic reconfiguration API

While databases deployed on one server are traditionally given exclusive access to the machine’s resources, they are generally also over-provisioned to handle peak load scenarios. Vela tries to avoid this by controlling the resources of each Data Processing Instance and the number of such instances in the system, adapting to the load. This makes the dynamic system reconfiguration a key aspect of Vela. For example, as the load on the system increases or decreases its replication factor is adjusted.

We differentiate between two types of reconfiguration operations: DPI level (fine grained) reconfiguration and System level (coarse grained) reconfiguration. At the DPI level, we can reconfigure the provisioned number of cores and the amount of memory. At the System level, we can reconfigure the number of DPIs as well as their placement in the cluster. The following reconfiguration API is used.

- `addCoresToDPI` and `removeCoresFromDPI` control the number of cores allocated to each DPI. They rely on Xen’s ability to hot plug cores in a VM. If increasing the number of cores can not be satisfied on the local physical
machine, a moveDPI operation needs be performed, moving the VM to a new physical machine that has the required free cores.

- increaseDPIMemory and decreaseDPIMemory control the amount of RAM allocated to each DPI. Increasing or decreasing a DPI's memory relies on Xen’s memory ballooning. Similar to adding cores, if there is insufficient free memory on the local physical machine, a moveDPI operation needs to be performed before.

- addDPI operation is used to spawn new DPIs. A new Master is added to the system for each new tenant, while a new Satellite can be added to mitigate an increase in the load of a tenant. Adding a new Master requires spawning a new VM and starting the Master database in it. Adding a Satellite is more involved. Besides spawning a new VM and starting a new database in this VM, Vela needs to ensure that the database is identical to the one in the Master. Instead of halting requests to the Master database and cloning it, Vela relies on the HotBackup.

- removeDPI takes out one DPI from the system. When a Master is removed, no more incoming requests are accepted. When all outstanding requests are served, the Master and all its Satellites are stopped (database then VM). The operation corresponds to removing a tenant from the system. In the case of stopping a Satellite, no more transactions are routed to it and once the outstanding ones complete, the database and the encapsulating VM are stopped and their resources (CPU and RAM) are freed.

- moveDPI operation handles moving a Data Processing Instance in the cluster from one physical machine to another physical machine. Two types of “move” operations can be performed. A cold move will make the Data Processing Instance unusable during the operation. This is implemented in Vela either by removeDPI followed by addDPI or through Xen’s cold migration. With hot moving, the Data Processing Instance is migrated while it is still serving request, relying on Xen’s live migration. Cold moves are faster then hot moves. As Satellites in Vela do not have any state associated, except for the currently running transactions, they can always be cold moved. Masters however can never be taken offline and consequently are always hot moved. The approach of live migration of virtualized databases presented in the Albatross system [36] could also be applied for the Master DPIs of Vela.
3.6 Target functions

The dynamic reconfiguration of Vela can be done manually or automatically based on monitored metrics. The system reports low level metrics for each Virtual Machine corresponding to each DPI (e.g., CPU utilizations, memory consumption and disk/network utilizations) as well as per-tenant statistics (e.g., overall throughput, read-only and update transaction latencies, etc.). The system maintains a set of user specified target functions that describe threshold values for monitored metrics and simple policies based on the reconfiguration API for achieving these thresholds. Basic tenant SLAs, like desired response time, can be expressed through target functions. In the evaluation of the system, we exemplify how the reconfiguration API and target functions work together.

For automatic reconfiguration support for target functions involving system response time, we rely on the Observer for determining the best response time of read-only workload. The Observer is an optional DPI, always deployed on 1 core. The Router ensures that if an Observer is present, it will always have one and only one transaction routed to it. As only one transaction is being executed at a time there is no contention in the Observer. Also as it runs on only 1 core it exhibits no scalability issues. The Observer approximates the best transaction latency for the current workload and reports it as a metric that can be used in target functions.

In order to simplify the definition of target functions, we introduce the concept of helper “check” functions. The role of the “check” functions is to report if a monitored metric is within a given tolerance from a target value. For a monitored metric $M$, the check function compares the value at the current time $t$ ($M_t$) with the target value.

In the case in which the user knows beforehand the target value ($T$), the check function returns:

$$V(M_t) > \frac{100 + \text{tolerance}}{100} \times T$$

In the case in which the target value is not known beforehand, but can be obtained from the Observer node (i.e., lower limit for response time), the check function returns:

$$V(M_t) > \frac{100 + \text{tolerance}}{100} \times O(T_t),$$

indicating if the monitored metric $M$ at time $t$ is matching the best attainable value reported by the Observer at time $t$ $O(T_t)$, within the given tolerance.
3.6. TARGET FUNCTIONS

Target functions are implemented in Vela through support for user-generated functions that act as control loops. They are designed to work as a feedback mechanism that monitors variables and compares them to a desired “target” (or setpoint) value. In this thesis we focus on showing what are the monitored variables and the system APIs that relate to them, and use simple control mechanism for achieving the desired target values. More advanced control loop models have been amply studied in classical control theory research. The type of control loops that we use are closed-loop. As we change the system state through a call to the reconfiguration API, we monitor its effect and feed that back into the control loop for the next evaluation.

We consider that for many use-cases, some of which we exemplify in Section 4.5.2, very naive control loops are sufficient. For implementing more complex target functions, the application of closed-loop transfer functions or of PID controllers might be interesting to pursue.

Our main goal was to enable support for target functions and identify the reconfiguration API and the monitored metrics. With such support in place, we were able to build target functions similar to the self-controlling model for software based on control theory. While not in the scope of this thesis, interesting follow-up points that would further strengthen the automatic reconfiguration capabilities of Vela relate to the responsiveness, stability and concurrent satisfiability of multiple target functions (which might be conflicting).

3.6.1 Automatic satellite scaling

Given a cluster of multicores we designed Vela such that it can scale both up and out. One trade-off has not been discussed so far is whether Vela should opt for few replicas with many cores (fat replicas) or for many replicas with few cores (thin replicas). Few fat replicas are easier to manage and require overall less RAM but will hit the scalability issues of individual databases. Many thin replicas increase the overall RAM consumption and transaction routing time.

Each Satellite should be run at the optimal size with respect to the number of cores. Unfortunately the sweet spot depends on the hardware, database software and workload.

Based on the our empirical experience, we present an algorithm that adjusts the number of cores in a Satellite. On one hand, it assumes that all Satellites are uniform and that the clients of each tenant have the same workload distribution. On the other hand it does tolerate changes in the workload, as long as all clients exhibit the same change.
CHAPTER 3. VELA: DESIGN AND IMPLEMENTATION

1. Software is accounted for as this is an online approach and is based on measurements taken at runtime.

2. Workload is accounted for using the Observer Data Processing Instance.

3. Hardware is accounted for in Satellite scaling through probing. If the check is not satisfied, the Satellite cores are increased as long as the response time improves. Once the optimal core count for a Satellite is determined, it is used to spawn new Satellites until the check is satisfied.

In refining the algorithm, we observed that the cache locality matters and that the cores that share the same cache behave similarly. The adapt function described in Figure 3.6 controls the process. Starting with one Satellite having one core, we expand the Satellite over all the cores sharing the same L2 cache (first package). If the required check is not met, we expand to the cores sharing the LLC cache and then to different NUMA nodes. When moving from one level to another, we “probe” to see if a speedup is gained or not (testNextPackage). If no performance gain is obtained, then we stop increasing the cores. Otherwise we go ahead and allocate all the cores of that level to the Satellite (fillPackage). Once the size of a Satellite is determined (the sweet-spot for the number of cores per Satellite), more Satellites of the same size are spawned (addSatellites) in an attempt to satisfy the check function. In Sections 4.2 and 4.3 we present a detailed evaluation of Vela’s scale-up ability that gave us the insights for this scaling algorithm.

3.7 System deployment examples

In this section we present a series of deployment examples and guidelines for Vela. The section covers both deployments on large multicores, clusters, and cloud infrastructures. The key aspects in deploying Vela relate to resource allocation for each DPI and for the Router. In the case of virtualization, we also have to account on each machine for the resources allocated to the Virtual Machine Monitor.

3.7.1 Single machine deployments

When deploying Vela on a single machine, all the system components share the machine. Figure 3.7 exemplifies a possible deployment of Vela on a 48 core machine with 8 sockets (each with 6 cores). The Router comprising the client facing interface and the dispatcher is placed on 4 cores of the same socket. It receives all the requests from the clients which are then dispatched to either Satellites or the
3.7. SYSTEM DEPLOYMENT EXAMPLES

```
TESTNEXTPACKAGE()
1  if TESTINCREASE(crtCores)
2   then crtCores ← crtCores + 1
3  else return FULL
4  if PERF_GAIN(crtCores)
5   then return OK
6  else crtCores ← crtCores - 1
7  return !OK

ADDSATELLITES()
1  while check(O, V) and haveHW()
2  do ADDSATELLITE_DPI()
3   if ! check(O, V)
4   then return DONE
5  return WARN(ProvisionHardware)

FILLPACKAGE()
1  while crtCores < coresInCrtPkg
2  do crtCores ← crtCores + 1
3  if ! check(O, V)
4  then return DONE
5  return !DONE

ADAPTPACKAGE()
1  if ! check(O, V)
2   then return DONE
3  repeat
4   if ! check(O, V)
5     then return DONE
6  until OK == TESTNEXTPACKAGE()
7  return ADDSATELLITES()
```

Figure 3.6: Automatic Satellite core count scaling algorithm

Master. The green lines represent the path for the queries coming through read-only transactions. The red lines show the path for the requests coming through an update transaction. All the WriteSets that are extracted from the Master are propagated to the Satellite databases.

The Master is allocated 8 cores while two Satellites have 12 cores and another
two have 6 cores. As we do not rely on virtualization for this deployment, we partition the resources at the OS level. Each Data Processing Instance is pinned to the cores it was allocated. This way there is no contention on the cores. The pinning is done using the Linux “numactl” tools and library. The processes are started with a “prefer local” NUMA policy, also specified through “numactl”. Alternatively, a “strict local” policy would ensure that memory is always local to the NUMA node to which the component is pinned, but can lead to bad memory
Moving to a setup in which the Data Processing Instances and the Router are encapsulated in Virtual Machines (presented in Figure 3.8), the resource allocation is no longer done through “numactl” mechanisms. The configuration of each VM holds the resources it controls (e.g., cores and main memory).

The advantage of encapsulating the DPIs and the Router in Virtual Machines comes from better support for resource reconfiguration – adding/removing cores or memory can be done through the Virtual Machine Monitor. For example in the case of “numactl”, the cores on which the system components are scheduled can be changed but this requires obtaining a list of all processes of each component and changing their affinity. In the case of PostgreSQL, which forks for each new connection, changing the affinity of each process is more involved, requiring changes on the main process, and then iterating over the many (possible hundreds) of child processes. With respect to memory, the amount allocated to each DPI can be controlled at a finer granularity than with “numactl” which only operates over whole NUMA regions.

The disadvantage of encapsulating system components in Virtual Machines comes from the additional indirection of the Virtual Machine Monitor (Xen VMM in Figure 3.8) and from additional resource overhead of multiple OS instances. The extent to which these affect overall performance and how we overcame these over-
heads are topics discussed in Chapter 6.

### 3.7.2 Cluster deployments

Both the virtualized and non-virtualized setups of Vela can be extended from a single machine to a cluster. Favoring the benefits of virtualization, when Vela is deployed on clusters its components are always virtualized.

![Deployment example illustrating the use of VMs over a cluster of multicores](image)

Figure 3.9: Deployment example illustrating the use of VMs over a cluster of multicores

Figure 3.9 illustrates the components of Vela, in a distributed deployment over a cluster of multicores. In the presented deployment Vela handles two tenants (A and B), each having a Master, a HotBackup and multiple Satellites. Tenant A also has an Observer. The setup spans multiple physical nodes, each with a different set of resources (cores and RAM). The System Model holds the layout of the whole system, mapping DPIs (and their encapsulating Virtual Machines) to the pool resource in the cluster.

Master nodes are preferably placed on separate cluster nodes, taking full advantage of local disk drives for the transaction logs, while storing the data on Network Attached Storage. All read-only DPIs (Satellites, HotBackups and Observers) keep the data in main-memory.

All three exemplified classes of possible deployments are used in the following chapter in the evaluation of the system.
3.7.3 Amazon EC2 deployments

We also investigated Amazon EC2 as a deployment infrastructure. We found two limitations in the existing infrastructure provided by Amazon EC2.

A first limitation was in the Xen Virtual Machines. As pointed out on support forums [103], IRQ balancing of network interrupts on multicore Virtual Machines is problematic, becoming a bottleneck for Vela’s Router which handles many fast I/O requests. In our local clusters we mitigated this using multiple bonded Virtual Network Interfaces (VIFs for short). In Amazon EC2 instances, an IRQ balance daemon along with manual configuration of Receive and Transmit Side Steering (software implementations of common hardware NIC features) partly solved the problem by spreading the load from one core.

A second limitation was the highly variable latency among large instances (in the same availability zone). Variable latencies between the Router and Satellites causes the WriteSets to be applied at different rates to the replicas. For the workloads that we investigated, the average request execution times are comparable to the network Round Trip Time. Consequently most read-only transactions cluster in Satellites “closer” to the Router, effectively impeding a scalability study.

While with some engineering the first limitation can be bypassed, the second limitation comes from a design in Amazon EC2 infrastructure that we can not solve. We argue that being able to obtain a pool of instances that have relatively similar network latencies should be a feature provided by the cloud provider. Amazon EC2 offers low-latency network between instances only for its High Performance Computing (HPC) Clusters. While Vela can scale on such a cluster, the pricing and diversity of hardware offered are still prohibitive.

3.8 Chapter summary

In this chapter we presented the design on Vela, a system based on off-the-shelf databases that has scalability, flexibility and multi-tenancy as design goals.

The system relies on primary-master replication providing snapshot isolation consistency guarantees. The chosen replication allows the system to scale at the same time over different machines, as well as within a single multicore machine. The system extends the basic primary-master replication scheme with specialized types of replicas for enhancing the functionality of the system (e.g.: adding new replicas at runtime or observing best achievable latencies).

The presented system is also flexible, exposing a reconfiguration API that can be manually or automatically controlled. Using features provided by virtualization,
Vela gains a uniform view of the resources it is deployed on, which it can dynamically reallocate between its components. Coarse reconfiguration operations allow the system to add or remove components in a deployment, while fine reconfiguration operations allow it to change the resources (CPU, memory) of its components. Using target functions, Vela monitors system metrics and can trigger the reconfiguration API such that service level agreements can be met as load in the system changes.

The benefits of virtualization are not always for free. Through careful engineering we show how Vela can take advantage of virtualization without paying a high price in terms of performance or functionality.

In the last section of the chapter we presented example system deployments in single multicore servers, over clusters and in Amazon EC2’s infrastructure. Following the deployments section, we proceed in the next chapter to evaluate the properties of the system, its performance, and its offered functionalities.
In this chapter we present the evaluation of Vela. Based on its design goals, we focus the evaluation on the following performance and functional aspects:

**Scaling up non-virtualized** off-the-shelf databases on a single large multicore machine. In this evaluation we emphasize that data replication within the boundaries of the same physical machine can outperform traditional stand-alone off-the-shelf databases, both in scalability and performance. The databases are run in a Linux environment, on bare metal – without virtualization. These series of experiments emphasize the feature subset that Vela inherited from the Multimed system.

**Scaling up virtualized** off-the-shelf databases on a single large multicore machine. This study has the purpose of showing the costs and benefits of encapsulating the Data Processing Instances of Vela in virtual machines. With virtualization, Vela gains flexibility with respect to online reconfiguration at a small price in the resource overhead.

**Scaling out** off-the-shelf databases over a cluster of multicore servers. In these evaluation we show that Vela can seamlessly scale within individual cluster nodes, as well as over the entire cluster. Virtualization is used for building a virtual cluster of resource nodes over the physical server cluster.
Online reconfiguration of the system can be performed at different levels of granularity. In this study, we show how Vela deployed over a virtual cluster of nodes can be reconfigured in runtime as changes in the offered load occur or as simple service level agreements need to be fulfilled.

Multitenancy in Vela is supported through its design, allowing multiple client databases to be concurrently managed within a single system deployment. This evaluation focuses on showing the performance isolation of the tenants and the ability to re-allocate resources among tenants at runtime.

### 4.1 Benchmarks overview

In the performance, scalability and reconfiguration evaluation of Vela, we have relied on workloads and datasets based on two standard benchmarks. The two benchmarks are TPC-W and TPC-E, both specified by the Transaction Processing Performance Council [122]. We have used our own implementations of the two benchmarks.

#### 4.1.1 TPC-W

The TPC-W [121] Benchmark emulates a web commerce system. It consists of a schema, dataset and workload specification for the data tier along with an application tier specification for supporting web commerce operations. All the interactions from the application tier to the data tier are transactional.

The TPC-W benchmark specifies three different workload mixes. The mixes consist of the same set of transactions but the occurrence ratios of each individual transaction is different. The three mixes are: Browsing (with 5% updates), Shopping (with 20% updates) and Ordering (with 50% updates). Out of these three, we focus on the Browsing and Shopping mixes. The Ordering mix is disk intensive and hits an I/O bottleneck before any proper CPU usage is seen.

The TPC-W benchmark specifies both an application and a database level. We implemented only the database level, as this is the point of interest for Vela. Due to the lack of the application level, some features required for correctly implementing the benchmark had to be emulated at the database level. For example the shopping cart, which should reside in the web server’s session state, is present in our implementation as a table in the database. In order to limit the side effects of holding the shopping cart in the database, an upper bound is placed on the number of entries that it can hold, equal to the maximum number of concurrent clients.
4.2 SCALING UP OVER BARE METAL

4.1.2 TPC-E

The TPC-E [118] Benchmark is used for testing database performance characteristics under a modern OLTP workload. The benchmark specifies the schema, data population and workload characteristics inspired from a brokerage firm. Even though the naming and scope of the workload is inspired from a brokerage firm, the benchmark creators intend it to be generic and representative for a wide variety of OLTP style workloads.

The benchmark defines clients that connect to their portfolios and perform operations on these. Besides client account keyed data accesses, the benchmark also defines a market driven set of interactions which can be used to adjust the update frequency.

4.2 Scaling up over bare metal

Regardless if virtualization is used or not, Vela offers a viable alternative to standalone database engines for harnessing the increasing number of cores in a multicore server. As we mentioned previously, Vela’s ability to better scale on multicores comes from treating the machine as a set of distributed resources rather than as a large parallel system. The fundamental difference between Vela and standalone database engines is that our system relies on replication rather than on parallelization. To better understand the performance and scalability implications of the two approaches in the context of multicores, we compare Vela with conventional database engines running on multicore machines. We measure the throughput and response time of each system while running on a different number of cores, clients, and different database sizes. We also characterize the overhead and applicability of Vela under different workloads. Aiming at a fair comparison between a traditional standalone database management system and Vela, we used the TPC-W benchmark, which allows us to quantify the behavior under different update loads.

We present the scalability of the four systems under study: standalone and Vela versions of PostgreSQL and MySQL, proving that indeed, Vela can take full advantage of the computational resources, as long as the workload is not disk bound. We also show how the four systems perform with increasing load and how Vela’s performance is boosted by the different configuration optimizations used. Finally, we show the overhead of Vela compared to standalone database instances.
4.2.1 Experimental setup, workloads and datasets

All the experiments were carried out on a four-socket AMD Opteron Processor 6174 with 48 cores, 128GB of RAM and two 146GB 15k RPM Seagate® Savvio® disks in RAID1.

Each CPU consists of two dies, with 6 cores per die. Each core has a local L1 (128KB) and L2 cache (512KB). Each die has a shared L3 cache (12MB). The dies within a CPU are connected with two HyperTransport (HT) links between each other, each one of them having two additional HT links.

For the experiments with a low number of Satellites (less than five), each Satellite was allocated entirely within a CPU, respectively within a die, to avoid competition for the cache. In the experiments with ten Satellites, partial replication was used, making the databases smaller. In this case, each Satellite was deployed on four cores for a total of three satellites per socket. Two of these Satellites are entirely within a die and the third spawns two dies within the same CPU. Due to the small size of the replicas (the point we want to make with partial replication), we have not encountered cache competition problems when Satellites share the L3 cache.

The hard disks in our machine prevented us from exploring more write intensive loads. Nevertheless, the features and behavior of Vela can be well studied in this hardware platform. A faster disk would only change at which point the Master hits the I/O bottleneck, improving the performance of Vela even further.

The operating system used is a 64-bit Ubuntu 10.04 LTS Server, running PostgreSQL 8.3, MySQL 5.1 and Sun Java SDK 1.6.

The workloads used are based on the TPC-W Benchmark. The datasets were populated according to the benchmark specification to size of 2GB and 20GB. Each experimental run consists of having clients connect to the database and issue transactions (read-only and update), as per the specifications of the TPC-W mix being run. Clients issue the load for a time period of 30 minutes, without think times.

Each experiment runs on a fresh copy of the database, so that dataset evolution does not affect the measurements. For consistent results, the configuration parameters (e.g., for memory and threading) of PostgreSQL and MySQL are fixed to the same values for both the standalone and Vela systems.

The clients are emulated by means of 10 physical machines. This way more than 1000 clients can load the target system without incurring overheads due to contention on the client side. Clients are implemented in Java and are used to emit the workload as well as to measure throughput and response time.

We have done extensive tests on Vela, trying to find the optimal configuration to use in the experiments. The number of cores on which the Satellites and the
4.2. SCALING UP OVER BARE METAL

Master are deployed can be adjusted. Also, the number of cores allocated for Vela’s Router code can be configured. In the experiments below we mention the number of Satellites (#S) and the optimization (C0-C2) that were used.

4.2.2 PostgreSQL: standalone vs. Vela

This section compares the performance of PostgreSQL and Vela running on top of PostgreSQL. Unless otherwise specified, the default optimization level for Vela is C1 - in which all Satellites are full data replicas in RAM.

Query intensive workload

Figures 4.1(a) and 4.1(b) present the scalability of PostgreSQL compared to Vela, in the case of the 2GB database, and 200 clients. The x-axis shows the number of cores used by both Vela and PostgreSQL, as well as the number of Satellites coordinated by Vela.

Both the throughput (Figure 4.1(a)) and the response time (Figure 4.1(b)) show that the TPC-W Browsing mix places a lot of pressure on standalone PostgreSQL, causing severe scalability problems with the number of cores. The Router running on 4 cores, the Master on 4 cores, and each Satellite on 4 cores scales up almost linearly to a total of 40 cores (equivalent of 8 Satellites). The limit is reached when the disk I/O bound is hit: all queries run extremely fast, leaving only update transactions in the system to run longer and face contention on the disk. The gap between the linear scalability line and Vela’s performance is constant, being caused by the computational resources required by the Router.

Figures 4.1(c) and 4.1(d) present the throughput of PostgreSQL (running on different numbers of cores) and of Vela (running with different configuration optimizations), as the number of clients increases. Note that PostgreSQL has problems in scaling with the number of clients issuing the workload, and its performance at 48 cores is lower than at 12.

For both dataset sizes, Vela (at all optimization levels) outperforms the standalone version of PostgreSQL. The C0 optimization level for Vela shows higher error bars, as all Satellites are going concurrently to disk, in order to persist updates. Switching to the C1 optimization level, we reduce the contention on disk by using more main memory. We see an improvement of more than 1000 transactions per second between the naïve C0 and optimized C1 versions of Vela. Finally, switching to the less generic optimization level C2, Vela accommodates even more Satellites in the available memory, and can take advantage of the available computational resources, until the disk I/O limit is hit. Using the C2 optimization, the
Figure 4.1: PostgreSQL standalone vs. Vela, running the TPC-W Browsing mix
problem of load interaction is also solved by routing the “heavy” (more analytical), queries to different Satellites, offloading the other Data Processing Instances in the system.

A highly remarkable feature of Vela is that it retains a steady behavior with increasing number of concurrent clients (up to 1000), without exhibiting performance degradation. Looking at the corresponding response times, even under heavy load, Vela’s response time is less than 1 second, indicating that it is not only solving the problems of load interaction, but also the client handling limitations of PostgreSQL.

Increased update workload

Figures 4.2(a) and 4.2(b) show that even in the case of the Shopping mix, PostgreSQL can not scale with the number of available cores, on the 2GB database, with 400 clients. Vela scales up to 16 cores (2 Satellites), at which point the disk becomes a bottleneck. Vela’s performance stays flat with increasing cores, while that of PostgreSQL drops.

Figures 4.2(c) and 4.2(d) show that PostgreSQL can not scale with the number of clients for this workload either, regardless of the database size. For the Shopping mix, standalone PostgreSQL’s performance is slightly better than for the Browsing mix due to the reduced number of heavy queries. In the case of Vela, for a small number of clients, all queries run very fast, leaving the updates to compete on the Master. Past 150 clients, the run time of queries increases and the contention in the Master is removed, allowing Vela to better use the available Satellites. We again observe that Vela’s behavior is steady and predictable with increasing load.

Using the C0 optimization level and for a low number of clients, Vela performs worse than PostgreSQL, especially on the 20GB database, although it is more stable as the number of clients increases. With more updates in the system and with all of the Satellites writing to disk Vela is blocked on I/O. As in the previous case, the C1 optimization solves the problem: standard deviation is reduced and the throughput increases. The C2 optimization at the same number of satellites also gives the system a performance gain as the WriteSets that need to be applied on the Satellites run faster.

Both in the case of a query intensive workload (Browsing mix) and in the case of increased update workload (Shopping mix), PostgreSQL does not scale with the number of cores nor with the number of clients, regardless of the database size. PostgreSQL’s inability to scale with the number of clients is due to the fact that for each new client a new process is spawned on the server. This might lead to the conclusion that the number of processes is far greater than what the operating
CHAPTER 4. VELA: SYSTEM EVALUATION

Figure 4.2: PostgreSQL standalone vs. Vela, running the TPC-W Shopping mix
system and the hardware can handle. This is disproved by Vela, which can cope with 1000 clients in spite of the limitations of PostgreSQL. The problem in this case is not the large number of processes in the system, but rather the inability of a single PostgreSQL engine to handle high concurrency. Since Vela splits the number of clients over a set of smaller sized Satellites, it reduces the contention in each engine, resulting in a higher throughput.

4.2.3 MySQL: standalone vs. Vela

In this section we compare standalone MySQL to Vela relying on MySQL database engines in each Data Processing Instance. For MySQL, the experiments were carried out using its InnoDB storage engine. This engine is the most stable and frequently used storage engine available for MySQL for transactional workloads. However, it has some peculiar characteristics: (i) it acts as a queuing system, allowing just a fixed number of concurrent threads to operate over the data (storage engine threads); (ii) it is slower than the PostgreSQL engine for disk operations (for the two versions compared here). In all the results presented below, the number of cores available for MySQL is equal to the number of storage engine threads. Being a queuing system, MySQL will not show a degradation in throughput with the number of clients, but rather exhibits linear increase in response time. For this reason, the experiments for MySQL only go up to 400 clients. Going beyond 400 clients would only further increase transaction response times which are already high at this number of clients.

Query intensive workload

Figures 4.3(a) and 4.3(b) present the ability of standalone MySQL, and of Vela, to scale with the amount of computational resources, in the case of the 2GB database and 200 clients. The x-axis, as before, indicates the total number of cores available for MySQL and Vela, as well as the number of Satellites coordinated by Vela. Each Satellite runs on 4 cores.

In the case of the TPC-W Browsing mix, we notice that MySQL, like PostgreSQL before it, does not scale with the number of cores. Figure 4.3(a) shows that MySQL performs best at 12 cores. Adding more cores increases contention and performance degrades.

The same conclusion can be seen in the throughput and response time plots for both the 2GB and 20GB datasets (Figures 4.3(e) and 4.3(d)), that show the performance of MySQL (running on different number of cores) and of Vela (running on different configurations and optimization levels) with increasing clients. Since
Figure 4.3: MySQL standalone vs. Vela, running the TPC-W Browsing mix
4.2. SCALING UP OVER BARE METAL

(a) Scalability throughput: 2GB, 200 clients  (b) Scalability response time: 2GB, 200 clients

(c) Throughput for 2GB database  (d) Throughput for 20GB database

(e) Response time for 2GB database  (f) Response time for 20GB database

Figure 4.4: MySQL standalone vs. Vela, running the TPC-W Shopping mix
the behavior is independent of the dataset, we conclude that the contention is not caused by a small dataset, but rather by the synchronization primitives (i.e., mutexes) that are used by MySQL throughout its entire code.

In contrast, Vela scales with the number of cores. Figure 4.3(a) shows that on the 2GB dataset, Vela scales up to 6 satellites, at which point the disk I/O becomes the bottleneck in the system, and the throughput and response times are flat. The fact that Vela on top of PostgreSQL scaled in the same test up to 8 satellites corroborates the fact the PostgreSQL’s storage engine is faster than MySQL’s InnoDB for this workload.

The three configurations and optimization levels that we have tested for Vela show that by replicating data, Vela can outperform standalone MySQL by a factor of 2, before it reaches the disk I/O bound. The C0 configuration shows a behavior similar to standalone MySQL’s best run. Removing the contention on disk by switching to the C1 configuration, the performance of Vela is further increased. The C2 optimization does not yield better performance than C1. The system is already disk bound and load interaction does not influence MySQL for this workload. To improve performance here, a faster disk or lower I/O latency is needed.

**Increased update workload**

The scalability plot (Figure 4.4(a)), shows that MySQL performs best at 8 cores. With more cores performance degrades, confirming that contention is the bottleneck, not disk I/O. Vela scales up to 16 cores, at which point the throughput flattens confirming that the disk becomes the bottleneck.

Figures 4.4(c) and 4.4(d) show that on larger datasets data contention decreases, allowing standalone MySQL to perform better. On the 2GB database, Vela brings an improvement of 3×. In the case of the 20GB database, it achieves a 1.5× improvement.

The error bars on the Vela line running a larger number of clients in Figure 4.4(d), show that its DPIs are not correctly balanced, producing an oscillating throughput. Fine tuning the Router would give a more uniform behavior of the system. The focus of this evaluation is not on micro-optimizing the system for maximum throughput, but rather to show that where designs based purely on parallelization do not easily scale to with the increasing number of cores present in large multicore machines, replication based solutions can easily be used.
4.2. SCALING UP OVER BARE METAL

4.2.4 Replication overhead

In order to understand how big is the overhead induced by the replication model of Vela and if replication changes in any way the load characteristics, we ran an experiment in which the system was configured to have no Satellites, just a Master that handles all transactions.

Figure 4.5 shows the throughput of Vela running a Master DPI on 6 cores and the Router on 2 cores, compared to the throughput of PostgreSQL running on 4 and 8 cores, for a 25GB database, proving that, as expected Vela running just a Master performs similar to plain PostgreSQL.

Figures 4.5(a) and 4.5(b) show the throughput and respectively the response time of PostgreSQL vs. Vela. The line for Vela (running a 6 core Master DPI) lays right in between that of a 4 core and 8 core PostgreSQL instance. In the case of Vela, 2 cores are allocated to the Router, yielding the overhead of our replication approach.

Taking a closer look at the response time breakdowns presented in Figures 4.5(c) and 4.5(d) it is clear that Vela is acting as proxy to the PostgreSQL instance in its DPI, without changing its characteristics. For each individual interaction defined in the TPC-W Benchmark we plot the average response time. The “U-” prefixed interactions are update interactions, while the “R-” prefixed ones are the read-only interactions. Within averaging error and accounting for the 2 core difference between the 8 core standalone PostgreSQL and Vela’s Master 6 core PostgreSQL instance we can conclude that the latency of individual transactions is the same.

4.2.5 Effects of load separation

In the motivation for a replicated based approach to scaling data processing on multicores, we mentioned load interaction as a cause of reduced scalability, besides contention on synchronization primitives. While we have shown how replication aids with synchronization contention, in this section we show how replication enables load separation as a solution to the load interaction problem.

Figure 4.6 shows the response times for PostgreSQL and Vela on a 25GB dataset. In all cases, the response times of Vela are smaller than those of PostgreSQL.

Each Satellite holds a replica containing a subset of the TPC-W defined tables item, author and order_line, being able to answer three TPC-W interactions (SearchRequest, SearchResults and BestSellers). These interactions represent 34% out of the full TPC-W Browsing mix. All other interactions are routed to the Master DPI which holds all the TPC-W defined tables.
Figures 4.6(a) and 4.6(b) provide a per-interaction response time break down. Taking a closer look at the response time breakdowns we can see the reason why Vela outperforms PostgreSQL: the response time for the read-only interactions is significantly lower. We can conclude that by offloading part of read-only transactions to Satellite DPIs, Vela improves their response time.

4.2.6 Standalone database improvements

We observed that in the case of PostgreSQL, recent versions have claimed improved scalability on multicore hardware. Consequently we investigate the improvement
4.2. SCALING UP OVER BARE METAL

Figure 4.6: Effect of load separation in Vela, running TPC-W Browsing mix

between the 8.3 version used in the previous experiments and the newer 9.2 version.

Figure 4.7 shows the throughput as the number of cores allocated to PostgreSQL is increased. A constant workload is synchronously, with no think time issued by 50 and 100 clients. The workload is based on the TPC-W benchmark specification.
(Browsing mix). In version 9.2 the performance no longer degrades as the number of cores is increased, and for a small number of clients it exhibits remarkable scalability. This shows that contention on the synchronization primitives (e.g., the `slock` function described in [18, 116]) was reduced as compared to earlier versions. The contention due to increased load on the other hand quickly degrades performance. By tuning PostgreSQL’s configuration (shared buffer, work memory, etc.) we were still unable to overcome this.

PostgreSQL 9.2 exhibits better scalability than previous versions, though as shown in Figure 4.7, this scalability improvements amount to little as the load increases: in the underload scenario there are less clients than cores in the machine, effectively leaving the cores underutilized. As soon as we move to more clients than cores, PostgreSQL stops scaling.
4.3 Scaling up with virtualization

For the scalability study of Vela on large multicore machines (scale-up) when relying on virtualization we used a 64 core (4 sockets × 16 cores) AMD Opteron 6276 clocked at 2.3GHz with 256GB DDR3 RAM clocked at 1333MHz. The machine was used both in a Native (Bare metal) setup (running a 64-bit Linux 3.6.6 on bare metal) and in a Virtualized setup, in which Xen 4.1.2 was used as a Virtual Machine Monitor (VMM) (with Xen’s Dom0 and all the paravirtualized DomUs running a 64-bit Linux 3.6.6).

As the database engines underlying each Data Processing Instance of Vela we opted for PostgreSQL 9.2. This off-the-shelf database is used both as a baseline in standalone experiments (running in a Native setup) and as the underlying database for Vela (running in a Virtualized setup). Workload wise, we chose two read-intensive workloads from the TPC-W and TPC-E benchmarks: TPC-W Browsing mix ≈ 95% reads and TPC-E ≈ 85% reads. Focusing on the data-processing system, we have omitted the application stack from the benchmark and connect the clients straight to the database engine.

In this section we present a series of experiments showing that Vela indeed scales on multicores under different read-intensive workloads. For this scale up study, datasets of ≈ 23GB for TPC-W Browsing and of ≈ 11GB for TPC-E were used. For the PostgreSQL setups the dataset was stored on Network Attached Storage (NAS), while the transaction logs were on a local SSD drive. In the case of Vela, the Master’s database was stored on the same NAS and the transaction logs on the same SSD drive as for standalone PostgreSQL. All Satellites held the dataset and the transaction logs in main-memory.

Figure 4.8 shows the scalability of Vela compared to that of standalone PostgreSQL. Four systems are studied: Native and Virtualized PostgreSQL, as well as Native and Virtualized Vela. The Native setups do not rely on virtualization. In these setups both PostgreSQL and Vela were run Linux on bare metal. The scalability trends exhibited by these lines match the findings detailed in the previous section. The Virtualized setups of both PostgreSQL and Vela follow, within error margins, the lines of the Native setups. This indicates that there is no significant performance or scalability overhead due to virtualization. This is expected as all the data is cached in main-memory for all setups, so we see no latency/read overheads from the NAS. Also, the TPC-W Browsing workload has very few updates, which creates little pressure on the local SSD that stores the transaction logs. For the rest of the analysis of the scalability of Vela in this section, we compare Native PostgreSQL with Virtualized Vela. While this is not an apples-to-apples comparison, it shows the extreme cases: Native PostgreSQL which runs in the setup for which it was designed (and has peak performance) and Virtualized Vela, which
might incur small performance penalties from virtualization. Even so, we show that Vela outperforms Native PostgreSQL.

For the throughput and response time plots in Figure 4.8, all four systems were under a load generated by 200 clients, each sending blocking requests, according to the TPC-W benchmark, in the Browsing mix. Under this workload Native
4.3. SCALING UP WITH VIRTUALIZATION

PostgreSQL has difficulties in scaling as we increase the number of cores it has at its disposal, for the same reasons discussed in the previous section. Increasing the cores by 8 each time, a slow increase in throughput is attained until 32 cores, point at which we observe NUMA effects. These lead to a performance drop, followed by a slow recovery as more cores are added.

Virtualized Vela’s scalability line starts at 22 cores: 2 cores have been allocated

Figure 4.9: Scaling up with number of cores, PostgreSQL standalone vs. Vela, running TPC-E
to Xen’s Dom0, 8 cores to Vela’s Router and 12 cores to the Master. All subsequent data-points correspond to adding additional Satellites, each assigned 8 cores. With the first Satellite, the throughput increases, as read load is offloaded to the Master (since the single Satellite has difficulties in keeping up with the incoming WriteSets). An incoming read-only transaction cannot be bound to a Satellite when none of the Satellites are up-to-date. This happens when they are overloaded and as a consequence the transaction is routed to the Master.

<table>
<thead>
<tr>
<th># Satellites</th>
<th>Unfulfilled Satellite bindings / second</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Not applicable</td>
</tr>
<tr>
<td>1</td>
<td>422.750</td>
</tr>
<tr>
<td>2</td>
<td>86.025</td>
</tr>
<tr>
<td>3</td>
<td>7.840</td>
</tr>
<tr>
<td>4</td>
<td>0.035</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 4.10:* Effect of overloaded Satellites in Vela, running TPC-WB

Figure 4.10 shows the average number of transactions per second that had to be bound to the Master because no Satellite was up-to-date. We observe that with only one Satellite more than 400 read-only transactions are routed to the Master per second. Once the second Satellite is added, performance does not increase, but fewer read-only transactions are being routed to the Master, and are handled by the two Satellites that can now keep up with the Master. With subsequent Satellites, throughput increases linearly up to the 5th point at which the Master’s CPU starts to become the next bottleneck.

<table>
<thead>
<tr>
<th># Satellites</th>
<th>Master Idle CPU %</th>
<th>Master Avg. # tx</th>
<th>Master Tx-log disk util. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
<td>193</td>
<td>1.39</td>
</tr>
<tr>
<td>1</td>
<td>10.87</td>
<td>45</td>
<td>4.40</td>
</tr>
<tr>
<td>2</td>
<td>68.19</td>
<td>4</td>
<td>5.28</td>
</tr>
<tr>
<td>3</td>
<td>75.00</td>
<td>5</td>
<td>6.32</td>
</tr>
<tr>
<td>4</td>
<td>72.49</td>
<td>9</td>
<td>11.65</td>
</tr>
<tr>
<td>5</td>
<td>17.57</td>
<td>34</td>
<td>16.92</td>
</tr>
</tbody>
</table>

*Figure 4.11:* Master load metrics vs. increasing # of Satellites in Vela, running TPC-WB

Figure 4.11 shows some of the Master’s load metrics, enabling us to understand when and why it is a bottleneck. With few Satellites, we remark that it’s CPU idle % is close to 0, which implies that the Master’s CPU is the bottleneck. As the number of Satellites increases, the total number of active transactions in the Master decreases and the idle CPU % increases, effectively removing this bottleneck.
More Satellites in the system result in the Master handling exclusively the update transactions. After adding the 4th and especially the 5th Satellite, we see that the Master’s CPU idle % starts to decrease while the active connections increase. As all the read-only transactions are handled by the Satellites as fast as possible, the update transactions appear to be more frequent (as their run time becomes comparably larger than that of the read-only ones). The fact that the Master has to handle more concurrent update transactions is supported by the increase in the transaction disk utilization.

Figure 4.9 shows the scalability of Virtualized Vela compared to that of Native PostgreSQL, when both systems were under a load generated by 200 clients, each sending blocking requests, according to the TPC-E benchmark. The workload is adapted (not having any Market Feed Emulation) such that a constant $\approx 15\%$ of transactions are update transactions, and the amount of data transmitted through the network is minimized. This allows us to evaluate the scalability of the data processing rather than the network traffic.

Native PostgreSQL scales linearly with the cores up to 32, though with high standard deviation. After 32 cores, its throughput slowly degrades with additional cores. As in the previous experiment, Virtualized Vela’s line starts at a larger number of cores – 24 in this case. This is due to the resources allocated to the management domain Dom0, to the Router and to the Master. Since the workload does put a lot of pressure on the network I/O with low latency requirements, we placed the Router on Xen’s Dom0, while keeping the Master and Satellites in DomUs. At 24 cores, the Router running in Dom0 has 8 cores, the Master is spawned on 8 cores and a Satellite is spawned on 8 cores. Since additional Satellites increase the throughput significantly, we alternate the addition of Satellites (8 cores each) with increasing the cores for the Router in Dom0 (by 8 cores), while leaving the Master at a constant 8 cores.

It is clear that Virtualized Vela has an added overhead of resources when deployed on a single machine, which is slightly larger than in the non-virtualized case. The cost of the Router, the Master and the Virtualization layer add up to the used cores and main-memory. Even so, Virtualized Vela outperforms a traditional database by reducing contention on synchronization and minimizing workload interaction. Vela also has a much better scaling trend with the number of cores.

4.4 Scaling out over cluster of multicores

For the scalability study over clusters of multicores (scale out), we used a cluster of 12 servers, each with 16 cores (Intel Xeon L5520 2.27GHz) with Hyperthreading
enabled, with 2 NUMA Nodes, and a total of 24GB RAM. Each cluster node runs Xen 4.1.2 with 2 cores and 2GB in Dom0 (Linux Kernel 3.6.6), leaving 14 cores and 22GB for DomUs to be spawned. The network connecting Vela’s components as well as the clients to the Router is a 1Gbit network. This experimental setup was also used for the elasticity and multi tenancy studies presented in the following sections.

![Figure 4.12](image_url)

**Figure 4.12:** Scaling out over a cluster of multicores, Vela running TPC-W *Browsing* mix
4.4. SCALING OUT OVER CLUSTER OF MULTICORES

Under the same workloads (TPC-W, in the Browsing mix and TPC-E) that were used for demonstrating Vela’s capability of scaling with the number of cores on a large server, we show that Vela can scale out until either the Master’s transaction log disk becomes the bottleneck or the network’s bandwidth is fully utilized.

Figure 4.12 shows Vela’s scalability trends as we increasingly add more Satellites, while blocking clients are issuing requests according to the TPC-W benchmark, in the Browsing mix. The dataset is ≈ 6GB.

All Satellites are configured to use 4 cores, and hold the data and the transaction logs in main memory. The Master is allocated 14 cores in a VM using up all the cores and memory of a cluster machine (16 total cores with 2 pinned to Dom0). The Master holds the data on Network Attached Storage with the transaction logs on a local SSD drive. The Master has the shared buffers large enough to cache all the data in main memory. The Router is running on a Native system (without virtualization) on an entire cluster node, in order to avoid the network issues described in Section 3.4.

As we increase the number of Satellites, we expect a linear increase in throughput. We observe in Figure 4.12 that for 50 and 100 clients, the system is underloaded. With 200 and 400 clients, the throughput is the same (lines completely overlap). The overlap in the throughput line correlated with the difference in the response times for the two lines indicate that the transaction log disk of the Master is the bottleneck. Monitoring the Master DPI during the experiment running on 25 Satellites, we recorded the CPU usage and the transaction log’s disk utilization.

<table>
<thead>
<tr>
<th># Clients</th>
<th>Master Idle CPU %</th>
<th>Master Avg. # tx</th>
<th>Master Tx-log disk util. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>34.79</td>
<td>21</td>
<td>72.95</td>
</tr>
<tr>
<td>200</td>
<td>21.41</td>
<td>83</td>
<td>94.41</td>
</tr>
<tr>
<td>400</td>
<td>9.33</td>
<td>251</td>
<td>95.33</td>
</tr>
<tr>
<td>400 (non durable)</td>
<td>0.89</td>
<td>50</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Figure 4.13: Master load metrics vs. increasing load over 25 Satellites in Vela, running TPC-WB

From the table in Figure 4.13 we observe that as the number of clients increases, so does the CPU usage and transaction disk utilization for the Master DPI (for the first 3 lines, which correspond to durable configurations). The direct cause for this is the increasing number of active transactions bound to the Master. Similar to the case of scaling up, as the read-only transactions are executed very fast (due to the large number of Satellites), the update transactions seem to agglomerate in the Master DPI, pushing its limits. With the current disk setup of the Master
DPI, the disk is fully saturated for both 200 and 400 clients. From the CPU idle % trend, we assume that if the disk bottleneck is removed, the CPU will be the next to follow.

![Graph showing throughput and response time with number of satellites.

Figure 4.14: Scaling out over a cluster of multicores, Vela running TPC-E]

To test this hypothesis and to further push the scalability of the system, we moved the Master’s transaction log in main memory – effectively giving up durability in the system. In this non-durable setup, with load offered by 400 clients, we
see a further increase in throughput. The flattening of the throughput line from 20 to 25 satellites is this time due to reaching a CPU bottleneck on the Master (see the 4th line in the table in Figure 4.13), which can only be solved by migrating it to a machine with more cores. Also, when running with 25 Satellites, the latency of the read workload cannot be improved and most of the 400 client connections end up being update transactions in the Master.

We point out an interesting artifact. That the flat throughput between 3 and 5 Satellites is caused by contention in the Satellites which cannot keep up with the Master. Consequently the Master also receives read-only transactions, improving the throughput.

Figure 4.14 shows Vela’s scalability as more Satellites are added, while 200 blocking clients are issuing requests according to the TPC-E benchmark. The dataset is $\approx 11\, \text{GB}$.

The transactions of the TPC-E benchmark yield on average 8KB of data that is sent over the network, quickly saturating its bandwidth. In our setup a 1Gbit network gets saturated quite fast. Using 14 core Satellites, the network is saturated after 2 Satellites, while with smaller 6 core Satellites, the saturation occurs after 4 Satellites. Peaking close to 10’000 transactions per second, with an average of 8KB per transaction, we are effectively transmitting $\approx 78\, \text{MB/second}$.

Scaling both up and out, Vela reaches high throughput rates over a commodity multicore cluster, processing more than 15’000 transactions/sec with latencies less than 50 milliseconds. With the given cluster nodes, 3 Satellites are collocated on each physical machine. We used 9 cluster nodes to spawn the 25 Satellites and one HotBackup node (for dynamically adding Satellites to the system). Another node was used exclusively for the Master and another node for the Router.

### 4.5 Online system reconfiguration

In this section we present a series of use-cases for manual and automatic online system reconfiguration. The focus of the evaluation is to demonstrate the functionality of the online-reconfiguration mechanisms based on the designed reconfiguration API. In contrast with the previous evaluations of Vela we do not emphasize performance or scalability.

All the experiments in this reconfiguration and elasticity study were carried out over the same cluster used in the scale out experiments. This section presents both manual and automatic online reconfiguration of Vela, under a constant load generated by 200 clients issuing requests based on the TPC-W benchmark, with the Browsing mix. While the reconfiguration API offers support for shrinking the
system, we present policies only for dynamically expanding the system, arguing the symmetry of the two types of operations.

### 4.5.1 Manual reconfiguration

In the manual reconfiguration experiment, Vela is monitored and reconfigured in runtime. The purpose of this experiment is to show that Data Processing Instance level reconfigurations can lead to improved performance. The experiment also shows that the reconfiguration needs to address correlations in the system in order to yield improvements.

Figure 4.15 shows the overall system throughput during the 50 minute experiment time, with samples taken every minute. The experiment starts with Vela configured with 4 Satellites and 1 Master as well as 1 HotBackup. The Satellites are in an “idle”-mode, having only 1 core. We increase their cores by 1 every 10 minutes, until we reach 4 cores per Satellite. A performance gain is seen by going from 1 to 2 cores (after 10 minutes), but not from 2 to 3 or 4 cores. The reason for this is that as the load is being handled by the Satellites increasingly faster, the
4.5. ONLINE SYSTEM RECONFIGURATION

Router becomes the bottleneck. Increasing the Router cores from 1 to 10 at the 40 minute mark further improves throughput, effectively enabling the Satellites to take full advantage of the extra cores.

Increasing the cores for the Satellites and for the Router is done using the reconfiguration API `addCoresToDPI`. The benefit of the extra cores is evident from the increases in throughput. The effectiveness of the `addCoresToDPI` API is also observable. The system can take advantage of the extra core almost instantly. The system does not take time to react to the new cores.

4.5.2 Target functions

Manually monitoring and reconfiguring the system is an interesting option for systems that observe rare changes in received load or requirements. Alternatively, knowing what are the possible bottlenecks, they can be automatically addressed using target functions. This removes the need for manual intervention. In this section we shall focus on showing the primitives that allow us to monitor and react to changes in the system load. With the use of target functions we enable defining a series of monitor-react loops. The focus of this evaluation is to show use-cases for the target functions, exemplifying the metrics that can be monitored and the set of actions that can be taken. As described in the previous chapter, we are focusing on showing what can be done with the online reconfiguration API, rather than on the coordination and scheduling of concurrent functions, or on the reaction speeds of these functions.

In the following we shall exemplify different system metrics that can be monitored and show how simple target functions can be written in order to react to changes in the monitored metrics, using Vela’s online reconfiguration API.

The target functions that we describe in this section follow a simple pattern, common to many loop-control functions. The target functions are active as long as the system is running, monitor the system state and if a target is not met, react through Vela’s reconfiguration API and enter a wait state allowing the changes to take effect.

The wait time between consecutive probes of the monitored state should balance the responsiveness to changes and measurement overhead. Depending on the frequencies of changes in the load (e.g., load increases or load characteristics change), the wait time can be adjusted. In our evaluations we did not investigate high-frequency load changes. While an upper limit can be arbitrarily set for the wait-time, the lower limit should be more carefully chosen. Too frequent measurements can add an overhead to a loaded system. Also, depending on the nature of the measurement, it might take time to complete. In the following evaluations,
the measurement time was capped to 5 seconds for obtaining OS metrics. This becomes the lowest limit for the wait time. On top of that, the wait time needs to be large enough to allow the system to react to any reconfiguration API invocations. For example if a DPI is allocated more memory, we allow time for the database engine to take advantage of the extra memory, which in most cases involves waiting for a cache to warm-up (in the order of minutes, depending on the memory size and speed of underlying durable storage). In another example, adding or removing cores from a DPI complete fast and the effect on the database engine is instantaneous (in the order of a few seconds). With these in mind, we opted for wait times in the range of 1-2 minutes as the measurement overhead is negligible at this rate and it allows the system to react to the reconfiguration API calls used in the exemplified target functions in this section.

**Adjusting the Router core count**

As the Router is CPU intensive, we set a target function on its CPU idle percentage. Vela monitors the CPU idle percentage on the Router and uses the reconfiguration API to adjust the system. The pseudo-code in Figure 4.16 illustrates the target function used in the experiment shown in Figure 4.17.

As long as the CPU idle percentage is less then 12%, \( \approx \frac{1}{8} \text{th} \) of the total CPU cycles, the Router core count is increased. If no more cores are available on the current cluster node, the Router is live-migrated on a cluster node that has more cores. The reconfiguration stops when the target function is reached or when the live migration cannot be done. The plot shows the system throughput on the primary y-axis, CPU-idle percentage on the secondary y-axis and the experiment time on the x-axis. The throughput grows as more cores are added to the Router. Once the target function of 12% CPU idle time is reached, the throughput stabilizes.

A set of questions need to be addressed here about the scalability of the Router. First, even give huge multicore machines, the Router will become the bottleneck at a certain point and it can not be migrated to a larger (i.e., more cores) machine. This happens when the Router manages many tenants in the system (which in turn leads to many concurrent requests without bottlenecking on any Master DPI). The solution in this case is to have a new Router that handles a subset of the tenants. Second, the network bandwidth of the Router can be the limiting factor. Either many small requests or fewer large requests passing through the Router can saturate the network bandwidth. As we optimized the system for latency and disabled Nagle’s algorithm [98] for TCP, small messages are not packed and lead to inefficient bandwidth utilization. The solutions include again having multiple Routers for subsets of tenants, or client-based load balancing [41] – effectively by-passing the Router for parts of the data I/O. Third, the memory consumption of
4.5. ONLINE SYSTEM RECONFIGURATION

TARGETFUNCTION()
1. $crtCores \leftarrow \text{getRouterCores}()$
2. $maxCores \leftarrow \text{getMaxRouterCores}()$
3. $targetCPUIdle \leftarrow 12$
4. 
5. while isSystemRunning()
6. do
7. $cpuIdle \leftarrow \text{getRouterCPUIdle}()$
8. if check($cpuIdle, targetCPUIdle$)
9. then
10. if $crtCores < maxCores$
11. then
12. $crtCores \leftarrow crtCores + 1$
13. setRouterCores($crtCores$)
14. else
15. if $OK == \text{liveMigrateRouter}()$
16. then
17. $maxCores \leftarrow \text{getMaxRouterCores}()$
18. 
19. SLEEP($w_{time}$)

FIGURE 4.16: Target function pseudo-code for Router CPU-idle %

the Router increases with the number of concurrent outstanding requests that it processes. Changing the Java VM heap size for the Router at runtime is not supported by existing Java Virtual Machines. Using Application Level Ballooning, a mechanism presented in Chapter 5 we offer a solution to this problem. Application Level Ballooning allows us to change the heap size of the Java VM running the Router as load increases and more time is wasted on frequent garbage collection.

Adjusting the Master memory

The Master DPI holds the dataset and the transaction log on persistent storage. It does this in order to offer durability in the system. For performance optimization, we want to cache as much of the data in main memory in order to speed up the transaction execution on the Master.

As PostgreSQL heavily relies on the Operating System’s buffer cache for storing the data files in RAM it is sufficient to hot plug memory in the Virtual Machine that encapsulates the Master DPI in order to cache more of the processed dataset.
Memory is added to the Master through the `increaseDPIMemory` API. Unfortunately this is not the general case with database engines. In the case of MySQL with the InnoDB storage engine for example we would have to resort to Application Level Ballooning for efficient online buffer pool size reconfiguration.

We setup the target function presented in Figure 4.18 to monitor the Master DPI database reads per second off the data storage medium. A direct consequence of increased memory for data caching translates in reduced reads from the persistent data storage medium. As long as the target of at most 100KB reads per second is not met for 10 consecutive checks (with 1 check per \(w_{time}\)), we increase the memory of the DPI by 100MB.

In the case of this target function, the \(w_{time}\) was set to 1 minute, such that it allows sufficient time for the memory allocation to complete and for the database to make use of its increased cache size.

Figure 4.19 shows the effect of the target function on the Master DPI. The Master is handling a 6GB dataset of the TPC-W benchmark. Initially the Master DPI has total of 2GB of memory. The large amount of reads/second (in the order of
4.5. ONLINE SYSTEM RECONFIGURATION

TARGETFUNCTION()
1  crtMem ← getMasterDPIMemory()
2  maxMem ← getDom0FreeMemeory()
3  targetReadsPerSec ← 100
4
5  while isSystemRunning()
6  do
7      dbReadsPerSec ← getMasterDPIReadsPerSec()
8      if check(dbReadPerSec, targetReadsPerSec)
9         then
10            if crtMem + 100 < maxMem
11               then
12                  INCREASEDPIMEMORY(crtMem + 100)
13                  crtMem ← crtMem + 100
14               else
15                  if OK == liveMigrateMaster()
16                     then
17                        maxMem ← getDom0FreeMemeory()
18                   end
19      end
20  end
21
22 Figure 4.18: Target function pseudo-code for Master DPI database reads/second

10MB/second) in the initial phase of the experiment indicate that the buffer cache accommodates little of the dataset causing the database engine to frequently read data from the persistent storage. As memory is gradually added to the system (at a rate of 100MB/minute) the amount of data read off persistent storage is reduced up to the point where it falls quickly below the target of 100KB/sec. After 27 minutes of experiment, the target is met (10 consecutive checks yielded an actual value less than 100KB read/sec). At this point the Master DPI has a total of $2GB + 27 \times 100MB = 4.7GB$ of main memory. While this is still below the dataset size of 6GB, it is sufficient to reduce the reads per second below the requested target as most of the frequently accessed data (indexes, fact tables, etc.) are in main memory. Increasing the memory at faster rates, rather than the slow 100MB/minute, would speed up the convergence process. The rate should be however less than the disk sequential read speed.
Adjusting the Satellite count

Target functions can also be set on monitored response variables that are specific to Vela, not only on generic system response variables like CPU, memory or disk. Figure 4.21 shows a target function set on the average response time of the system. The primary y-axis shows the throughput and the secondary y-axis the response time. The x-axis is the experiment time. A target value of 100 milliseconds for the average response time is set. The pseudo-code describes how this is achieved: as long as the target response time is not reached, a new satellite will be spawned through the `addDPI` API. The pseudo-code for the used target function is shown in Figure 4.20.

The experiment presented in Figure 4.21 starts with Vela configured with 1 Satellite of 4 cores. After 2 minutes a new Satellite is added to the system. Within less than 4 minutes the new Satellite receives load. This continues until the 4th Satellite is added, point at which the response time falls under the target value (minute 15 onwards).

In this target function the $w_{time}$ was set to 2 minutes, sufficient time for the
TARGETFUNCTION()
1  
crtSats ← getSatCount()
2  
targetRt ← 100
3  
4 while isSystemRunning()
5  
6  rt ← getAvgResponseTime()
7  
8  if check(rt, targetRt)
9  then
10    crtSats ← crtSats + 1
11    addSatellite()
12  
13 SLEEP(wtime)

Figure 4.20: Target function pseudo-code for the system’s response time

newly added satellite to warmup and catch up with the Master node, such that
the benefit of the added Satellite is observable.

In this approach a cut-off value was supplied. This value is unfortunately work-
load dependent. Using the Observer DPI described in Section 3.6, we monitor
the best attainable response time for the current hardware, software and workload
combination. The target for Vela can be set to a percentage of this value, making
the system agnostic of the workload. We denote the optimal response time con-
tinuously measured in the Observer as \( O(RT_t) \). Assuming that the Satellites and
the workload are uniform, we probe the response time from a Satellite to get the
runtime value \( V(RT_t) \) at time \( t \). A check function returns:

\[
V(RT_t) > \frac{100 + tolerance}{100} \times O(RT_t),
\]

indicating if the overall system response time is matching the attainable response
time, with the given tolerance.

As the latency of the read-only part of the workload is reduced, by adding
more Satellites to the system (approaching the optimal value \( O(RT) \)), the update
transactions “appear” to be more frequent and congest in the Master. The update
transactions in the Master translate into increased CPU usage and transaction
log pressure. The increased CPU usage is addressed in Vela by supporting online
addition of cores and live-migration. The pressure on the transaction log of the
database underlying the Master DPI manifests itself by both increased CPU wait
times (synchronous writes to the log) and peaking disk utilization. We present two different approaches to mitigate this problem. First, the write latency of transaction commits can be reduced through software-verification and virtualization enabled mechanisms like those explored in the RapiLog system [55]. Second, the amount of synchronous commits handled by the system can also be improved through a combination of RAID and SSD storage (i.e., improving the hardware).

4.6 Multitenancy support

Scaling and dynamic reconfiguration in Vela work on a per-tenant base. In this section we show that multiple Masters DPIs (each with its set of Satellites) can be configured in the same deployment of Vela. The dynamic reconfiguration plays an even larger role in this case, with Satellites being scaled for each tenant.

Figure 4.22 shows the individual response time for each of the four tenants deployed in Vela. Each has its own Master and 1 Satellite with 1 core. Each
4.6. MULTITENANCY SUPPORT

The fast drop in response time captures the effect of increasing the Satellite cores from 1 to 4. This is insufficient to meet the latency target and Vela adds a new 4 core Satellite for the tenant (which takes about 5 minutes). At this point the target latency is met and the system stabilizes. Under similar load conditions the tenants behave similarly and in isolation. Moreover, Vela can use the set of cluster nodes that it manages to dynamically add Satellites for any of the tenants.

Figure 4.23 shows Vela managing 20 tenant databases, each under a constant load from 40 clients (again, TPC-W, in the Browsing mix). All Master DPIs are collocated on the same cluster node and each tenant has 1 Satellite. Within standard deviation, all tenants perform similarly. The variation in response time is caused by contention on the transaction log storage of the cluster node handling the Master Data Processing Instances.

Figure 4.22: Per-tenant automatic system reconfiguration in a multi-tenant deployment
Figure 4.23: Performance isolation in a 20 tenant deployment with 40 clients issuing load to each tenant.
4.7 Chapter summary

In the evaluation of Vela we have empirically shown that the system full-fills its design goals.

First, the system scales with the size of the hardware that it has at its disposal. We showed that where off-the-shelf databases fall short in terms of scalability, Vela can do much better and scale linearly, offering the same consistency and durability guarantees as the standalone databases. Even though counter-intuitively, the extra latency of managing Satellites, keeping them updated and dispatching requests and responses through the Router is more than compensated by reducing synchronization contention and load interaction. The price such an approach is the increased RAM consumption for storing replicas in main-memory.

Second, through virtualization, the system can apply the same replication model to scale data processing over a cluster of multicores, until a hardware limit is reached. The limits of the system arise from either the transaction log of the Master or the network bandwidth of the Router, both well studied and understood problems. For the read-intensive workloads for which the system is designed, these limits only occur when the system is processing requests at very high rates (above 15,000 transactions per second), which is more than that claimed for SQL Azure [129] – in the order of few thousands of requests per second.

Third, the system provides an online reconfiguration API that can be used either manually or automatically through target functions. Through these mechanisms the system can be tuned to auto-scale as load increases in order to meet client requirements.

Fourth, by supporting multiple Masters, each with its subset of replicas, Vela offers multi-tenancy. This is useful for collocating clients, or as a form of data-partitioning for the same client when Vela does not need to support transactions across partitions.
Application Level Ballooning

Running systems software like database or language runtimes in virtualized environments poses a series of challenges. In Chapter 2 we discussed both possible performance overheads and functional miss-matches as such types of challenges.

In this chapter we focus on a functional mismatch between the possibilities of dynamic memory reconfiguration offered by virtualization and systems that do their own memory management. In this category we identify systems like most database engines or the language runtime systems. Specifically, we consider MySQL/InnoDB as a database engine and the OpenJDK Java Virtual Machine as a language runtime.

Both in the design and evaluation of Vela, we treat dynamic memory reconfiguration of both Data Processing Instances and the Router as an important requirement. Unfortunately, in most scenarios, the databases backing up the Data Processing Instances and Java VM supporting the Router are limited in their ability to dynamically reconfigure their memory footprint at runtime. For adding support to databases and language runtimes for dynamic memory reconfiguration we developed a novel mechanism which proves to be a generic solution to the problem. We refer to this mechanism as Application Level Ballooning (ALB).

This chapter further motivates the need for dynamic memory reconfiguration for virtualized systems, presents the limitations of current solutions, and introduces Application Level Ballooning along with its design and implementation details.
Finally, we evaluate the applicability, generality and performance of Application Level Ballooning in the context of MySQL/InnoDB and OpenJDK.

5.1 Overview and background

Virtualization in cloud computing and server consolidation enables applications previously deployed on dedicated machines to share a physical server, reducing resource consumption, energy, and space costs among other benefits. Vela perfectly exemplifies these benefits.

Statistically multiplexing server software and ensuring application performance as load changes, however, requires careful coordination policies. Besides this, virtual machine monitor (VMM) mechanisms like live migration and memory ballooning are used to dynamically reallocate resources, such as RAM, in virtual machines (VMs) to meet performance goals.

5.1.1 Reconfiguring memory allocation

Effective application consolidation though virtualization requires the ability to change the size of machine memory available to an application. We discuss four possible solutions to this problem.

**Restart:** We can stop the application and restart it with a larger memory pool backed by the Virtual Machine Monitor. For example, a database would stop accepting queries, wait for existing queries to complete, shut down, and restart with a larger memory pool. It requires no changes to the database, Operating System, or Virtual Machine Monitor, but entails significant downtime and hence reduced availability. The window of unavailability can be reduced by aborting all transactions when the database shuts down, and restarting them when it is rebooted. Unfortunately this approach may degrade overall throughput since useful work has now been thrown away and must be repeated. In the case of JVMs application restart might be optimized [62, 24], but restart is still a poor fit for high availability scenarios.

**Paging:** The Operating System can demand-page application memory to disk, perhaps in response to Virtual Machine Monitor memory ballooning. The problem is that server applications like databases pin memory to RAM precisely to avoid OS level paging. Like restart, paging can reduce an application’s RAM requirements
without code changes. However, paging undermines the optimizations performed
by a database or garbage collector: it becomes impossible to intelligently manage
database buffers, for example. In the worst case, double paging occurs: the
application touches a page in its memory in order to evict it and write it to disk,
unaware that that page has already been paged out. The page is needlessly read
in from disk (evicting another Virtual Machine page in the process), only to be
written back and discarded.

Rewrite: Applications can be rewritten to rely on the Operating System for re-
source management, no longer assuming a fixed memory pool. We have explored
some of these ideas in the COD system [45] that argues a better integration of
systems software (like databases) and the Operating System through a System
Knowledge Base and constraint solver. Long-term, we believe this will happen:
it is increasingly commercially important for a database to share a machine with
other applications, for example, and designing a database in this way is a promising
research topic. Short-term, however, the engineering cost is prohibitive. Databases
and language runtimes represent decades of careful engineering, with many fea-
ture interdependencies tied to many critical legacy applications, and a radically
redesigned storage manager or garbage collector for modern hardware (physical
and virtual) will take years to mature.

Conventional ballooning: A guest Operating System itself shares the char-
acteristics of our example applications, and Virtual Machine Monitor designers
rejected restart, paging, and rewrite as ways to vary the machine memory allo-
cated to a guest Operating System, in favor of memory ballooning [127]: each
guest kernel incorporates a balloon driver, typically a kernel module, which com-
municates with the Virtual Machine Monitor via a private channel. To the guest
Operating System, the balloon driver appears to allocate and deallocate pinned
pages of physical memory\footnote{Henceforth, we use the customary terminology: physical memory refers to a Virtual Ma-
chine’s memory that the guest Operating System manages, and machine memory denotes real
RAM managed by the Virtual Machine Memory.}. To reduce the memory used by a Virtual Machine, the
driver “inflates the balloon”, acquiring physical memory from the kernel’s alloca-
tor much as a device driver would allocate pinned DMA buffers. These pages are
now considered by the guest Operating System to be private to the balloon driver,
which notifies the Virtual Machine Monitor of the pages it has acquired. The Vir-
tual Machine Monitor converts the physical addresses to machine addresses, and
reclains these pages for use by other Virtual Machines. Conversely, “deflating the
balloon” returns pages from the Virtual Machine Monitor to the balloon driver,
which then “frees” them in the guest Operating System. Compared with the Virtual Machine Monitor transparently paging the guest, ballooning dramatically improves performance for applications whose memory is managed by the Operating System. Inflating the balloon increases memory pressure inside the guest, making it to page its own memory to virtual disk, thereby making a more informed page replacement choice than the Virtual Machine Monitor can. Unfortunately, this approach is not suitable for applications that manage their own memory, as the application is not involved and this approach becomes identical in behavior to Paging.

5.1.2 Requirements for ALB

Application Level Ballooning enables efficient memory ballooning for applications that manage their own memory. It achieves this by inserting a balloon module which allocates memory from the application’s pool and returns it to the Operating System via a system call, and extending the Operating System to return pages to the application. The *virtual* memory used by the application can therefore be dynamically varied between a minimal level and the full configured size.

However, while attractive, ALB’s effectiveness depends on some key assumptions. We discuss them below, and justify them with the aid of results from the evaluation of ALB, presented in Section 5.3.

First, available memory at runtime must **correlate with performance** for most workloads. For most workloads that we analyzed in the context of MySQL/InnoDB and OpenJDK we have seen a clear “knee” in the curve of performance vs. memory. There is a threshold memory size above which the system always performs well, while anything below that leads to fast performance degradation. Figure 5.1(a) shows such an example in the case of OpenJDK. Running an XML based benchmark (XMark) we monitor the total time for completing the submitted workload, while varying the Heap Size of the Java VM. Each measurement is broken down total time spent by the runtime doing actual work and the time spent on garbage collection tasks. The trend is clear: as the heap size is increased the time spent on garbage collection decreases while the actual useful work time stays constant. The XML document processed had a size of 1GB which translates to \( \approx 4 \text{GB} \) of DOM main memory data structures (the XML toolkit \([72]\) indicates that on average XML documents take up \(4 \times\) more memory for their representation). This is why we do not go with the heap size lower than 5GB.

Similar observations hold true in the case of MySQL/InnoDB, but due to different reasons. Figure 5.1(b) shows the average response time for a subset of queries of the TPC-H benchmark, while varying the configured database memory. With
5.1. OVERVIEW AND BACKGROUND

(a) Response time vs. JVM heap size

(b) Response time vs. configured DBMS memory

(c) Effect of conventional ballooning on query performance (TPC-H, Q19)

Figure 5.1: Supporting the requirements of ALB
varying degrees, queries run faster as the database memory is increased. More memory means for opportunities for caching data from disk. The queries that can benefit from more cached data will have shorter response times.

Second, ALB should not violate application assumptions about available memory. For example, a database query planner might base its optimization decisions on the statically configured memory size, whereas in reality much of that space has been ballooned out, causing worse performance than a static configuration with less memory. In the case of conventional ballooning this does not hold. Figure 5.1(c) compares the response time of a query from the TPC-H benchmark ran on MySQL statically configured (Static) with 4, 6, and 8 GB of memory to the case in which we forcefully reduced the memory available to MySQL through conventional ballooning (Dynamic). What happens in this case is that the database becomes delusional about where its data resides. We show in the evaluation of ALB that our approach does not suffer from this symptom.

Third, the engineering effort to modify the targeted application, the guest Operating System, and the Virtual Machine Monitor must not outweigh the benefits. We quantify the (small) code changes needed to implement our solution in Section 5.2.2.

Fourth, applications should respond quickly to changes in memory size due to ballooning. Databases heavily caching data require time to “warm up” newly available memory for buffers, whereas a Java VM can immediately allocate objects from new heap space. In Section 5.3.6 we evaluate our applications’ responsiveness to changes in memory allocation.

Finally, ALB as currently designed is not suitable for all applications. Some, such as PostgreSQL, delegate their memory management to the Operating System (via the file system buffer cache), making conventional ballooning sufficient and rendering ALB redundant. Others, such as the Tomcat application server, implement yet another layer of memory management above the Java VM, requiring an additional layer of ballooning.

5.2 ALB design

ALB enables the reallocation of machine memory (RAM) between memory managers of different applications (databases and Java VMs) residing in different virtual machines, while preserving the performance of each as a function of its currently available memory.

We implemented ALB for MySQL and OpenJDK, using a Linux guest Operating System and Xen [130]. We exploit the existing balloon functionality in Xen, and
extend the applications to support ballooning pages to and from MySQL’s buffer pool and OpenJDK’s heap. We modified the existing balloon driver in Linux as it was only designed to balloon pages from the kernel free list. Furthermore, ALB refers to ballooned pages by virtual addresses, requiring additional address translation.

### 5.2.1 Memory ballooning in Xen

Memory ballooning is not a novel concept. Xen, for example, provides an implementation of ballooning for moving memory between guest OSes. Xen maintains an area of shared memory for communication between the VMM and a guest OS, and this area includes the current “target memory” size for each VM. A memory balloon kernel module in each VM monitors the target memory value and reacts when it changes: when the target decreases, the kernel module will increase the balloon reservation, and when it increases, the balloon driver will decrease the balloon reservation.

At boot time, the balloon driver in each VM starts by inflating the balloon over a number of VM physical memory pages. Then it enters a “sleep” state, waiting for a change in the target memory.

When the balloon needs to be increased, the required number of physical pages is computed. For each one, the balloon driver requests a new physical page from the kernel’s LRU free list. All obtained pages are clean, free and unused memory. The balloon driver records their addresses in a kernel list of ballooned-out-pages and zeros out their contents. Next, the pages’ physical page numbers (i.e., physical addresses from the guest’s perspective) are translated into machine page numbers (the actual physical page numbers from the VMM perspective). Finally, the list of machine page numbers is passed to the VMM through a “hyper call”. The VMM then adds them to its LRU free list. Once the pages are in the VMM’s list of free pages they can be allocated to any VM, by decreasing the recipient VM’s balloon reservation.

When a balloon needs to be decreased, the balloon driver computes the number of memory pages required. It sets up a descriptor of ballooned out pages satisfying the request. The descriptor is passed to the VMM via a “hyper call”. In turn, the VMM populates (backs-up) the specified physical pages with free machine pages, and if this succeeds, the balloon driver updates the mappings of physical pages to machine pages and the page table in the guest OS. Finally, the new pages are cleaned and then freed in the guest OS (i.e., adding them to the VM’s LRU free list).
5.2.2 Application ballooning on top of Xen

As shown in Figure 5.2, the balloon module in ALB-aware applications interfaces with both the kernel and the management system (in our case this is Vela). The former occurs via a system call which frees or reclaims memory for the application’s balloon module. While we could have overloaded the `mmap` and `munmap` system calls to achieve this, we chose to add a new, orthogonal system call to allow flexibility in the OS, and to support applications which do not allocate memory at startup using `mmap`.

The management system determines policy, and conveys a target balloon size to the ALB module via an RPC call to a local socket. Applications which already have user-facing interfaces can also support other interfaces – for example, we also added SQL ballooning extensions to MySQL for testing.

Figure 5.3 shows ALB in operation. The management system controls the ballooning process by changing the balloon target for the application (1). The application allocates pages (2) and notifies the OS via a system call (3). The modified guest OS balloon driver processes the requests (4) and makes the memory changes visible to Xen with a hypercall (5).

The cost of implementing ALB is modest. Our approach required 870 new lines to the Linux guest kernel, most of which are in Xen’s memory balloon driver. In the case of InnoDB and MySQL, 229 lines of code were added, evenly distributed between implementing the ALB driver in the InnoDB heap and adding SQL language support in MySQL for the ALB manipulation. Finally, for OpenJDK we added 427 lines of code, most of which were in the garbage collector where the ALB driver is plugged in.

The small amount of changes that were done to the two application memory managers to support ALB indicate that implementing application level memory ballooning drivers is a minimal-effort task, as most of the complexity is abstracted in the kernel changes and the existing kernel-level balloon driver.

Following, we discuss in more detail the modifications required to MySQL/InnoDB, OpenJDK, and Xen/Linux to make ALB work.
5.2. ALB DESIGN

Management and Monitoring System
Manual or Automatic

(1) Notify Application of ALB Operation

Application
Memory Manager with ALB support

(2) increase/decrease Appl. balloon

User
Kernel

(3) Syscall for ALB MemOp

Kernel Balloon Driver

(4) increase/decrease Kernel balloon

VM1

(5) Hypercall for MemOp

VMM

Hardware

Application
Memory Manager with ALB support

User
Kernel

Kernel Balloon Driver

VM2

Figure 5.3: System architecture and call stack
5.2.3 The MySQL ALB module

While there are many choices of control channels for MySQL, we chose to extend the query parser with two new SQL commands:

\[
\text{balloon\_grow} | \text{balloon\_shrink} < \text{engine} > < \text{numberofpages} >
\]

These invoke the corresponding functions we added to MySQL’s `handlerton`\(^2\) storage manager API, and implemented in the InnoDB storage manager inside MySQL.

Our balloon module for MySQL is built into the InnoDB storage back-end. InnoDB manages its own memory, creating a fixed-size buffer pool at startup divided into 16kB InnoDB “pages” and is used for both the data cache and the lock table. Most pages are cache, managed with an LRU list.

The balloon module uses the internal `mem_heap_create_in_buffer` and `mem_heap_free` calls to acquire and release InnoDB pages. Once acquired, a page will not be used by the pool for caching data and is effectively removed from the available memory until freed.

To inflate the balloon, ALB acquires memory from the pool and adds it to a list of ballooned InnoDB pages. There is one such list per database instance. A list of the aligned virtual address ranges for pages is passed via the new system call to Linux for VM ballooning. To deflate the balloon, ALB traverses the list of ballooned pages to be returned to the LRU free list and notifies the kernel of the number of pages required. The pages are then faulted in by zeroing out the memory at the corresponding virtual addresses.

There are some subtle implementation issues. Figure 5.4 shows the structure of an InnoDB page: 16kB InnoDB pages consist of four 4kB virtual pages. InnoDB stores book-keeping metadata in the first of these 4kB pages when the InnoDB page is on the free list. A naïve ballooning module would therefore only be able to return 75% of the virtual pages to the OS. The first 4kB virtual page of the 16kB InnoDB page cannot be ballooned as InnoDB needs to be able to access the metadata fragment for traversing the free list.

Our implementation of the ballooning module in MySQL/InnoDB includes an optimization that copies the metadata (less than 300 bytes) from the first 4KB virtual page into a new, pre-allocated area of memory. This requires about 300MB extra memory to balloon 16GB. The net result is increasing utilization to

\[
\frac{16384MB - 300MB}{16384MB} \times 100 \approx 98\%
\]

\(^2\)The interface available in MySQL for implementing external storage managers.
5.2. ALB DESIGN

5.2.4 The OpenJDK ALB module

While offering support for dynamic memory reconfiguration of databases in virtualized environments is self-explanatory in the context of Vela, applying it to language runtimes like OpenJDK is not that obvious. We decided to pursue the design, implementation and evaluation of the OpenJDK ALB module as part of Vela in order to enhance the memory flexibility to Vela’s Router DPI (which is implemented as a Java application). An aside consequence of implementing the OpenJDK ALB module is the fact that we show how ALB is widely applicable to different classes of applications that run in virtualized environments and handle their own memory management.

The ALB module for OpenJDK is more involved than that for MySQL. While InnoDB uses most of the memory for caching, the Java VM passes most of its memory to the Java heap controlled by the garbage collector. In our approach we opted for a low overhead solution that shows all the benefits of application level ballooning for OpenJDK. We focus on ballooning in and out of the heap space, and modify the Parallel Scavenge Garbage Collector (PSGC) that is bundled with OpenJDK 7. At the end of this section we present our thoughts on alternative approaches for the placement of the ALB driver.

The Parallel Scavenge Garbage Collector is a generational collector. Figure 5.5 shows how this garbage collector partitions the heap space. The permanent generation holds class metadata. The young generation holds recently-created objects, and those that have survived a few garbage collections. Objects that survive longer are moved to the old generation. The young generation is also split
### Figure 5.5: OpenJDK parallel scavenger heap structure showing memory partitioning in different generations

Spaces in PSGC are bounded by start and end addresses, and all allocated memory sits between the *start address* and the *top address*. The top address is incremented as more objects are allocated. Collection compacts the space between the start and top addresses, removing holes and when necessary moving objects to other spaces.

We implemented support for Java VM ballooning through a new *balloon* space in the *young generation*. The balloon space can grow and increase the pressure on the *eden* space when necessary, or conversely contract and reduce the pressure. Our current implementation only balloons the *eden* space and does not support PSGC’s adaptive size policy, though the design does not prevent us supporting more spaces/generations and their adaptive resizing. There are no limiting factors in the design that impede support for adaptive resizing.

Enhancing the young generation with the *balloon* space was not the only possible choice. The *balloon* could enhance the old generation, or both young and old generation. There is a clear trade-off between where the ALB driver is placed and the type of workload that could benefit from it. For example, in a recent white paper [63] the authors study the impact of GC tuning for specific components of a Map-Reduce system. They identified that for the TeraSort workload that was investigated, the Map phase was putting a lot of pressure on the old generation,
alternating every $\approx 1$ minute from heavy usage to low usage of the heap space. Conversely, the Reduce phase puts more pressure on the young generation where a lot of objects are frequently created and deleted. For the Map phase, the system would benefit most from an ALB driver operating on the old generation, while in the case of the Reduce phase an ALB driver operating on the young generation would be beneficial. The life-time of the objects in the system is the one factor that influences the choice of the best heap space to the involved in the ballooning process. Many short lived objects put pressure on the young generation, while longer lived objects get tenured to the old generation.

A more completely engineered solution would add an ALB driver to both the old and young generation, with both being almost identical. In our case we opted for an implementation targeting the young generation as the workloads that we have investigated (i.e., XML document processing and querying) have proved to benefit mostly from such a design, having many short-lived objects appearing and being removed from the young generation.

In order to resize the spaces composing the heap, we need to compact them and prevent any other operations during ballooning. For this reason, the ballooning operation is performed at the same time as a full garbage collection. Before completing a full collection, we perform all outstanding ballooning operations. This means that the cost for ballooning operations in the JVM is influenced by the time needed to perform a garbage collection, as detailed in Section 5.3.6. A tighter coupling of the GC implementation with ALB would reduce much of this overhead.

Besides the balloon reservation mechanism, a new Java VM thread monitors incoming requests for balloon operations and invokes the garbage collector to perform a compaction followed by the resizing of the balloon space.

For an ALB driver to work, it needs to be able to make a reservation (that can be grown and shrunk) over the application managed memory area. The ALB driver needs to guarantee that the virtual address space that is backing up the reserved application memory will not change for the duration of the reservation. In the JVM we can make the reservation either over the virtual address space that backs up the heap, or over the object address space. For working over the virtual address space, the ALB driver interacts with the garbage collector because this is the component that make the allocation of memory from the OS. For working over the object address space, the ALB driver interacts with the object reference-to-virtual address mapping system.

If we opt to operate at the garbage collector level (i.e., virtual address space), then we have the benefits of working at a large granularity, we are independent of the object abstraction and the implementation is straight forward as most garbage
collectors already have the concept of heap partitioning. The downside is that the implementation is garbage collector dependent.

If we opt to operate the object reference to virtual address mapping level (i.e., object reference space), then we have the benefit of being independent of the garbage collector. The downside to the approach is handling all the changes in the mapping caused by object relocation and memory compaction (in order to avoid thrashing) which are done by most garbage collector implementations.

We considered alternative, garbage-collector-agnostic solutions which simply create special Java objects to inflate the balloon, as in a recent patent from VMware [78]. Unfortunately, this approach has problems:

- First, translating from an object reference to an OS virtual address, without modifying the Java VM, requires calling into native code via JNI \(^3\) (since the object address is not available within Java).

- Second, most garbage collectors move objects among and within heap spaces, requiring re-translation of the object address after each collection phase. OpenJDK does offer a runtime callback for garbage collection completion, but it would still result in considerable runtime overhead in remapping pages. Alternatively, objects used for memory reservation could be pinned in memory using JNI, though the implications of this vary between GC implementations. The description of the JNI calls for pinning memory (for instance represented as native arrays) [65] allows for either creating copies of the array (undesirable for memory ballooning) or actually pinning the memory for the duration that the JNI thread is executing between “GetPrimitiveArrayCritical” and “ReleasePrimitiveArrayCritical”. The JVM documentation remarks that for this duration the Garbage Collection might not be scheduled.

A third option might also be possible, but might require a larger re-design of the memory management in the JVM. By separating the memory allocation and resizing from the object tracking and compaction roles of the garbage collector, the ALB driver could be independent of the garbage collector but still operate on the virtual address space. Such an approach would also have the benefit that all heap operations (adaptive resizing, growth, etc.) would be handled together with the ALB memory reservation.

---

\(^3\)JNI is the Java Native Interface – a framework that enables Java applications to have components natively built for specific platforms. Such components can take advantage of platform specific functionality and have direct access to the system memory.
Consequently, without a larger redesign it is not possible to be fully independent of the garbage collector. The design we adopted does modify the PSGC, but it does not depend on JNI or runtime callbacks, it does not require address translations, and it is not impacted by regular garbage collector operations, thereby reducing the code complexity of the balloon mechanism.

5.2.5 Changes to the Linux kernel

To support ALB, we modified a Xen-patched Linux kernel so that it can be notified of ALB operations via a new system call, which required relatively little code. This modified Linux kernel is only used in the guest domains. The new system call takes as parameters (i) the ALB operation, (ii) the start of the virtual memory address and (iii) the size of the memory on which to operate. A pseudo-device or /proc file system entry would work as alternatives to our system call approach.

The ALB operation either inflates or deflates the balloon. The system call handles an inflate request by ensuring the candidate virtual pages refer to resident physical pages by faulting them in. Once resident, the physical page numbers are obtained, and a request to increase the kernel balloon is enqueued with Xen’s balloon driver. This operation is synchronous: a semaphore is set before placing the request on Xen’s request queue, then waiting for it to be unset by the dequeuing handler function. The kernel handles a deflate request by taking the number of OS pages required and enqueuing them with the kernel balloon driver. On return, Linux’s free list will have expanded to support subsequent use of the deflated application pages.

5.2.6 Changes to the Xen balloon driver

Xen’s balloon driver processes operations via a kernel work queue. For ALB operations, work items are enqueued as a result of executing the ballooning system call. Again, the changes to the Xen Balloon Driver are only needed in the guest.

A balloon_process function dequeues work items and either increases or decreases the kernel balloon. The same function handles both traditional ballooning requests and ALB requests. The main difference lies in the source of the physical pages that are ballooned. In one case these pages are allocated from or freed to the Linux kernel, while in the case of ALB the physical pages are taken from and returned to the corresponding application’s virtual memory area.

Both InnoDB and OpenJDK allocate their memory pool through an mmap operation. The requested memory is anonymous and private (meaning that it can
be swapped, is not backed up by any file descriptor, and is zero-filled). Virtual addresses from the application might or might not have been faulted in the guest OS. For ballooning out a page, we have to ensure that the physical page backing a virtual address is faulted in: we use the `get_user_pages` function \[^77\] for this. Faulting in the pages will add entries in the guest OS’s page table. Based on the `mmap` request, the physical pages obtained from the application are anonymous, private, with a map count of 1, and with a valid mapping pointer. Also they are present in both the page table of the guest OS and the kernel’s LRU list.

In contrast, the pages that are used by Xen for traditional ballooning are newly allocated in the kernel through a call to `alloc_pages` \[^77\]. Besides having another state & flags, they are not in the kernel’s LRU list. These pages are not anonymous, have no mapping (a map count of 0), and are not present in any page table.

Xen’s `hypervisor` for memory operations require that the pages moved from the guest OS into the Virtual Machine Monitor be in the same state as those obtained from a call to `alloc_pages`. This means that pages backing up an application’s `mmap`-ed memory require extrication for use by the balloon driver.

Since pages that we can obtain from the application are not in that state, a design choice needs to be made. Out of the space of possible solutions, we explored two: (i) updating pages and (ii) freeing and reallocating pages.

**Updating pages** is the more complex task of the two, but results in better performance. For this reason we chose this approach in our implementation.

When increasing the balloon, for each page that backs an application virtual address, we wait for the LRU to drain (i.e., for the page to be faulted in), clean its contents, move it from the LRU list to the ballooned list, and clear its mappings and anonymous state. Once the page is “sanitized” the page table is updated so that the virtual address no longer maps to this page. Next, the virtualization mapping between page-frame-numbers (PFNs) and machine-frame-numbers (MFNs) is removed. Finally, a `hypercall` notifies the Virtual Machine Monitor that the page is free and the guest OS has released control of that memory.

When decreasing the balloon, a buffer that will be populated by the Virtual Machine Monitor is initialized with pages removed from the list of ballooned-out pages. Once reclaimed through a `hypercall`, these pages are mapped in the page table at the correct application virtual addresses (these can then be re-enabled in the buffer pool / heap). The map count, page count and anonymous mappings are re-established, as well as the virtual memory page protections. Once this is done, the page is linked back into the kernel’s LRU list.
Freeing and reallocating pages is less complex than updating pages, and somewhat slower than updating pages.

When increasing the balloon, the page table entry matching the virtual addresses of the application page that will be ballooned out is cleared. Next, we “sanitize” the page and return it to the kernel’s list of free pages and drain the CPU list. Then, we proceed as with a normal OS ballooning operation, requesting pages from the kernel via alloc_page for each page that we need to balloon. When all necessary pages are obtained, their page table entries are cleared in the guest OS, the PFN-to-MFN mappings are invalidated, and the pages are added to the Ballooned-out list (without having to wait for a CPU drain operation). Finally, through a hypercall taking as parameter the list of PFNs, the memory is marked as free in the Virtual Machine Monitor, ready to be allocated to any guest OS.

When decreasing the balloon, the steps are the same as for normal ballooning. For each page needed to be returned to the application, we extract a page from the ballooned-out list, “sanitize” it and then return it to the kernel’s LRU free list [77]. No page table entry mappings need to be changed. When the application touches the memory, the pages will be faulted in.

5.3 Experimental evaluation

The aim of the experimental evaluation is fivefold:

- We compare the performance of a conventionally sized (i.e. statically provisioned) application with that of the same application sized by shrinking using ALB. We expect that both MySQL and the Java VM will have the same performance in the Ballooning case (i.e., shrunk to a specific size using ballooning) as in the Conventional case (statically configured to this smaller size).

- We show the performance characteristics over time of a system with two tenant applications, where ALB is used to dynamically reallocate memory between them.

- We examine a simple two-tier architecture, which relies on MySQL for the data tier and on the Java VM for the business tier. In this setup, we move memory during runtime between the data tier and the business tier, in order to reduce the overall latency of requests to the system.
We show a setup with four collocated database that can react to workload changes, mitigating the performance degradation caused by spikes in the workload through ALB.

We measure the performance of balloon inflation and deflation in both the database and the Java VM.

5.3.1 Experimental setup

We used a server with $2 \times$ Quad Core AMD Opteron 2376 2.3GHz Processors, 16GB of DDR2 RAM, and two 7200RPM 1TB SATAII drives for the overhead, in-flight and end-to-end experiments presented in Sections 5.3.2, 5.3.3, and respectively 5.3.4. A 64 Core AMD Opteron 6276 with 256GB of DDR3 RAM and 4 $\times$ OCZ Vertex 4, 256GB SATAIII SSD drives server was used for the database collocation experiment presented in Section 5.3.5.

We used Xen 4.1 [130] as the Virtual Machine Monitor, with 64-bit Ubuntu 10.04 (Linux kernel version 2.6.32) with Xen patches running in Dom0 as the management OS. The guest OSes were paravirtualized instances of 64-bit CentOS 6.2 (Linux kernel 2.6.32), with the VMs configured with 4 virtual cores each. The database engine was 64-bit MySQL 5.5 with the InnoDB storage engine. The Java VM was a 64-bit OpenJDK 7. Both MySQL and OpenJDK were compiled with gcc 4.4.3. Xen and the management Dom0 boot live images via PXE Boot and run from an in-memory file-system, removing possible disk interference caused by Xen itself from our experiments.

On these platforms we evaluated the characteristics of database Application Level Ballooning using queries from the TPC-H Benchmark [119] (for a dataset with a scale factor of 5, corresponding to approximately 5GB of raw data). For evaluating the Java VM Application Level Ballooning we used the XMark benchmark (for an XML dataset of 1GB) relying on the Nux toolkit [72] with Saxon 8 [105].

5.3.2 Overhead of ballooning

In order to understand the overhead of ALB, we investigate the performance of MySQL and OpenJDK under the following two scenarios:

Conventional: MySQL and OpenJDK run without any changes in a guest OS. The guest runs a paravirtualized Linux kernel, with the default Xen patches. While
the guest OS binaries run from a RAMdisk, the MySQL database files and the XMark input XML file are stored directly on a dedicated real disk. The amount of memory allocated to the InnoDB buffer pool is varied between 4 and 10GB by directly changing MySQL’s configuration file. The amount of heap space allocated to the Java VM is varied between 5 and 10GB. These provide the performance baselines for statically provisioned MySQL and OpenJDK instances.

**Ballooned:** MySQL and OpenJDK run with their individual ALB modifications in place, within a paravirtualized Linux guest as above. Additionally, the guest OS now contains our modifications to Linux and the balloon driver to support ALB. Again, the guest OS binaries are on a RAMdisk, and the data files are stored directly on a dedicated real disk. However, in this case the amount of memory available for the InnoDB buffer pool and the Java VM’s Heap are directly controlled via ALB. For each of the memory sizes above, we start MySQL and OpenJDK statically provisioned with the maximum memory size of 10G, and then balloon out pages in order to reach the desired size. We then measure the performance with the reduced amount of memory.

Due to the large dataset size (TPC-H with a scale factor of 5) and our storage setup, run time for some of the TPC-H queries was prohibitively long. We removed from our experiments queries that failed to complete within 30 minutes. Consequently, we present results for 17 queries out of 22. These are queries 1–4, 6, 8, 10, 11, 13–17, and 19–22. For the XMark queries, we omit running queries 8 through 12. These 5 queries are large “join” queries that failed to complete within 2 hours (on the 1GB input XML data file). Each XMark query is run 10 times in all the experiments, and we report the total time.

While the omitted queries dramatically extend the duration of each experiment, they behave similarly and so do not affect the results significantly.

**MySQL overhead**

Figure 5.6 shows the average response time of each query, in the two scenarios (Conventional and Ballooned), for two memory configurations: 4GB and 8GB. Figure 5.6(a) shows that the ballooned configuration performs almost identically to the conventional configuration for an I/O intensive setup, where most of the data can not be cached in the buffer pool. Similarly, Figure 5.6(b) shows that there is almost no overhead between the Ballooned and Conventional configurations for CPU intensive setups where most of the data is cached in the buffer pool.

From Figure 5.6 we conclude that there are no substantial differences between the two configurations we investigated. The differences we do see are caused by
CHAPTER 5. APPLICATION LEVEL BALLOONING

Figure 5.6: Ballooning overhead in MySQL: Conventional vs. Ballooned TPC-H query response times
randomness in the query parameters and non-deterministic variation in the system state. One major source of such randomness is MySQL’s random probing method used for cardinality estimation. The query optimizer in MySQL relies on estimated cardinalities of the indexes in order to determine which index should be used in the execution plan. The cardinalities are estimated using a method called “random dives” that, as specified by its name, randomly chooses a set of pages from the index and uses the values found in those pages to estimated the cardinality of the attribute. While a useful technique, the lack of control over the seed value for the random dives and the number of pages over which it is performed lead us to observe, quite often, different execution plans even for the same query with the same parameters. The InnoDB method `page_cwr_fcg_prng` uses a linear congruential pseudo-random number generator that relies on the current time as seed. A quick solution in which the seed value is fixed to a constant (instead of the current time) could prevent this. Regardless, we did not implement this change as we consider it would weaken the original design of the random dives method by possibly trading performance in favor of determinism.

Most of the queries in TPC-H benefit from a larger buffer pool. Also, most of the queries have an exponential-looking increase in response time with a linear decrease in the buffer pool size, with a clear threshold. The exceptions are those aggregation queries that require large table scans (e.g., query Q1). In some cases, we find that ballooning can lead to more complex effects on the performance of the database engine. Section 5.5 touches on this topic.

**OpenJDK overhead**

Figure 5.7 shows total runtime for the sequence of 15 XQueries in the XMark benchmark. Each bar depicts total run time, divided into time the Java VM did useful work in answering the XQueries and the time the Java VM spent on garbage collection (due to the heap size). In all experiments the workload stayed CPU bound. We draw two conclusions.

- First, increasing the Java VM heap size reduces the time overhead of garbage collection, thus speeding up query response time.

- Second, for all of the five memory configurations ranging from 5 to 10GB for the heap size, the performance of the ballooned configuration (“ALB”) closely resembles that of the conventional static configuration (“Static”).

We have found that the Java VM’s garbage collector that we used in our experiments is performing very many runtime optimizations, mostly based on heuristics.
In order to be able to compare results, we have fixed the old generation to 3GB, the “to” and “from” spaces of the young generation to 256MB each, leaving the “eden” space of the young generation as the only variable one. For achieving this we also had to turn off the Adaptive Size Policy of OpenJDK 7.

5.3.3 In-flight memory resizing

We also examined the performance characteristics of using ALB to reallocate memory between two virtual machines. In one experiment we resize the buffer pool of MySQL/InnoDB, while in the other we resize the heap size of OpenJDK. The experiment is carried out similarly for both systems, with two clients issuing the application specific workload to stress the application in the corresponding virtual machine.

Resizing MySQL’s buffer pools

For running the memory reallocation between two MySQL instances, we first bring up a database (VM1-DB) in one virtual machine configured to a total of 9GB of RAM, 8 of which reserved for the InnoDB buffer pool. We then reduce this allocation by 4GB using Application Level Ballooning, allowing us to bring up another database (VM2-DB) in the other virtual machine, also configured to a total of 9GB with 8 reserved for InnoDB.
5.3. EXPERIMENTAL EVALUATION

The workload used for this experiment consisted of the TPC-H queries 2, 3, 6, 11, 16, 17, 19, 20, and 22, selected because they have lower response times and allow us to present the throughput of the system as a function of time without having to aggregate over long periods.

During a 6 hour test, we changed the buffer pool size of the InnoDB storage engines every 2 hours by reallocating 4GB of memory back and forth between the two databases using ALB. After 120 minutes, we reduce the buffer pool of VM2-DB by 4GB and increase that of VM1-DB by the same amount. After 240 minutes, we reverse the process.

We refer to these two operations as in-flight resizing as they were performed while the databases were processing queries.

Figure 5.8 shows the throughput of the two systems as a function of time. The throughput is reported every 2 minutes. When the in-flight resizing operations occur, the throughput of the two systems change. At minute 120 we observe a rapid drop in throughput for VM2-DB (a large number of ballooned-out InnoDB pages will lead to more disk access for answering queries). At the same time, a steady increase in the throughput of VM2-DB is observed.

The gradual increase in throughput for VM2-DB is caused by the database warm-up period. It takes time for the new pages that were ballooned-in to the buffer pool to be actively used. Here we are performing a somewhat extreme experiment in which we move 4GB of memory, meaning that 50% of the buffer pool needs to be warmed up to reach steady-state performance. The same behavior can be observed for the in-flight resizing performed at minute 240.

Two features of the results stand out: the oscillating throughput and the apparently long delay before the database can fully exploit newly ballooned-in pages.

The oscillating throughput is a consequence of the nature of the workload. The queries used in the workload have very different response times. The dataset is too large for main memory, forcing InnoDB to frequently access the disk. This
behavior prevents us from running a large number of clients in order to observe a more stable throughput, as disk contention from multiple clients would make this experiment prohibitively slow. Also, the wide variation in response time across different queries (see Figure 5.1(b)) leads us to aggregate the throughput values over bins of 2 minutes.

The apparently long warm-up times (minutes 120-to-130 for VM1-DB and 240-to-250 for VM2-DB) are caused by the cache warm up time. The queries running at the point when memory is ballooned-in determine how fast the buffer pool is populated. For reference, the same warm-up phenomenon is seen at time 0, when the database starts with an empty data cache.

**Resizing the JVM heap**

As in the previous experiment we start the VMs sequentially. The first VM, configured to a total of 8GB, out of which we allocate 7GB to VM1-JVM. These 7GB are split into the different parts of the heap: a fixed 3GB old generation space, and a fixed 4GB for the young generation, including a 256MB to-space and a 256MB from-space, leaving 3.5GB for the eden space. We continue expanding the JVM balloon over 2GB from the eden space. With the newly released memory, we can start the second VM, giving it the same total memory of 8GB, out of which we allocate 7GB to VM2-JVM (again with 3GB for the old generation, 3.5GB for eden space and 256MB for to-space and 256MB for from-space).

Similar to the MySQL in-flight-resizing experiment, we run a 6 hour test, during which we reallocate 2GB of memory between the heaps (eden-space) of the two JVMs every 2 hours. The results are presented in Figure 5.9. At minutes 120 and 240 we notice how the throughputs of the two JVMs change. When the eden spaces increase from 1.5GB to 3.5GB (minute 120 for VM1-JVM and minute 240 for VM2-JVM), we see an instant increase in throughput. Comparing to the case
5.3. EXPERIMENTAL EVALUATION

of buffer pool of MySQL, the JVM can instantly make use of the extra memory without needing to warm up a cache.

5.3.4 An end-to-end example

ALB is not restricted to reallocating memory between the same type of application. In this experiment we move memory between MySQL and OpenJDK. The motivation is a simple two tier architecture. The data tier implemented through a MySQL database runs in a virtual machine and the business tier implemented as a Java application that processes XQueries runs in another virtual machine. The communication with the clients is done via a minimal socket-based interface for receiving queries and sending results back.

The synthetic workload that drives this benchmark requires that for each request a set of database queries needs to be run in the data-tier. Based on their results, a series of XML document processing operations will be performed in the business tier, and the results sent back to the client.

In the synthetic workload, for each client request, we perform a series of database queries (we ran the TPC-H queries 2, 3, 6, 11, 16, 17, 19, 20 and 22) on a TPC-H dataset with a scale factor of 5 (corresponding to approximately 5GB of raw data), followed by a series of XMark queries (1-7 and 13-20) on a 1GB input XML file. The response sent back to the client is the total runtime for each individual query.

The experiment starts by creating a virtual machine with a total of 9GB of RAM, in which we start VM1-DB, a MySQL instance with a buffer pool of 7.5GB. We then balloon 1.5GB out from the database, reducing its actual buffer pool to 6GB. Next we start a second virtual machine with a total of 7GB of RAM, in which we start VM2-JVM with a fixed heap of 6GB (3GB old generation, 2.5GB eden-space and the remaining 0.5GB equally split among the to/from spaces).

The initial configuration Static in the table in Figure 5.10, reports results for processing the workload without performing any memory movement between VM1-DB and VM2-JVM. VM1-DB runs with an actual buffer pool of 6GB and VM2-JVM runs with a fixed heap of 6GB.

The second configuration Dynamic in the table in Figure 5.10 starts in the same configuration as the Static one, but reallocates 1.5GB from VM2-JVM to VM1-DB before performing any database queries, and then moves the 1.5GB back to VM2-JVM before performing any XML processing.

Figure 5.10 presents the runtime break down for the two system configurations. For the Static configuration we do not spend any time on doing ballooning operations. In the Dynamic configuration we see that at the cost of 2 ballooning operations (each taking approximately 21 seconds) an overall improvement of $\approx 4x$
CHAPTER 5. APPLICATION LEVEL BALLOONING

<table>
<thead>
<tr>
<th>Config</th>
<th>DB Queries (sec)</th>
<th>XML Queries (sec)</th>
<th>Balloon Ops (sec)</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>2077.25</td>
<td>455.97</td>
<td>0</td>
<td>2533.22</td>
</tr>
<tr>
<td>Dynamic</td>
<td>164.57</td>
<td>450.37</td>
<td>21.53</td>
<td>638.00</td>
</tr>
</tbody>
</table>

**Figure 5.10:** Improving the end-to-end latency in a two-tier system: Static vs. Dynamic configurations

is gained. This approach of moving memory between the data and business layer only makes sense if the gain in query response latency covers the cost of the ballooning operations. Our current implementation of ALB is not necessarily suitable for faster interactive systems, where the overall latencies should fall well under 5 seconds, though we are confident that through engineering efforts this latency can be reduced.

5.3.5 Collocating database servers

**Figure 5.11:** Reacting to workload changes in a collocated database setup

The previous evaluations of Application Level Ballooning have proven its ability to re-allocate memory between databases and language runtimes, yielding performance benefits. We now present a larger scale experiment that highlights the
5.3. EXPERIMENTAL EVALUATION

benefits of ALB in the context of collocated database servers, having direct applicability in the Vela system.

For the experiment we used a 64 core, 256GB RAM AMD machine, running Xen: Dom0 on 16 cores with 2GB of RAM and 4 VMs, each having 12 cores and 36GB of RAM. Within each VM a MySQL/InnoDB database operates on a TPC-E dataset of 105GB (TPC-E scale factor of 5000 customers). The TPC-E dataset consists of brokerage transactions corresponding to different customers and the workload comprises mostly fast running read-transactions. All 4 databases have an initial InnoDB buffer pool of 8GB which can be increased up to 24GB through ALB.

The experiment consists of running a base workload (10 client threads, targeting 1000 of the 5000 TPC-E customers in the dataset) for a duration of 5 hours. An extra workload (20 client threads, targeting any of the 5000 customers in the dataset) increases the load on a database server. When a server receives extra load, the ALB will be triggered to increase the size of the buffer pool on the pressured database. Increasing the buffer pool mitigates the drop in performance due to the extra load. Once the extra load ceases, we shrink the buffer pool.

Figure 5.11 shows the throughput of the 4 databases, each under a constant base load and spurious extra loads. At the 30 minute mark the first database starts receiving the extra load. Due to pollution of the buffer pool (more data touched by the extra clients), there is an abrupt drop in throughput (top line in Figure 5.11).

A simple monitoring system (like that of Vela) detects the extra load and the drop in throughput and reacts by increasing the buffer pool of DB1 from 8GB to 16GB. Within 5 minutes, the throughput of DB1 recovers. After a total run of 40 minutes, the extra workload stops. When the extra workload stops (minute 70), the performance drops due to the reduced offered load and because the buffer pool still holds pages that are irrelevant for the base load. Within 2 minutes, the throughput stabilizes again and the monitoring system initiates the shrinking of the buffer size of DB1 back to 8GB (minute 75). The second drop in throughput is caused by the shrinking of the buffer pool. Finally, at minute 80 minute the throughput of DB1 is stabilized. Extra workload events happen at minutes 90, 150 and 210 for DB2, DB3 and DB4 respectively. The behavior is the same as in the case of DB1.

This shows how ALB can be used in real scenarios of database collocation in virtualized environments and how spikes in load can be handled on the fly, without stopping the database, just by increasing the size of the database buffer pools.

The reaction time to increasing the InnoDB buffer pool is determined by two factors: query diversity and the I/O subsystem. A high number of concurrent
queries will lead to a fast cache population. Higher throughput from the I/O subsystem also speeds up cache population. In the current experiment we found that the time required to warm-up the pages in the InnoDB buffer pool is bound by the I/O subsystem. Doubling the number of clients, the 90 percentile response time increases for some transactions by as much as 100% with an average of 62%. With 10 clients, the 90 percentile of response times of the transactions was in the range of 30 milliseconds to 350 milliseconds while with 20 clients it was from 30 milliseconds to 580 milliseconds. This means that more load might decrease the cache warm-up time, but would increase latency. Investigating the I/O subsystem, we saw that disks with fast seek times improve disk throughput (the TPC-E workload does many random data accesses [29]). For this collocation experiment each database was stored on a OCZ Vortex 4, 256GB SSD drive, which can yield 80k I/Os per second, for throughputs of up to 500MB/s (with no seek overhead).

Using these SSD drives (instead of traditional spinning disks), the bottleneck shifted from the disk drives to Xen’s Virtual Machine Monitor, where all I/O interrupts for the Virtual Machines were handled only on one core (having 100% utilization). As in the evaluation of Vela, balancing the interrupts among the VMM’s 16 cores can further reduce the time required to warm up the database’s buffer pools. We have found these types of performance limitations quite frequently in our performance evaluations of virtualized setups and discuss them further in this thesis in the dedicated Chapter 6.

5.3.6 ALB performance

We evaluate the performance of individual ALB operations by measuring the time it takes to increase or decrease the balloon, from the application level, by a certain number of pages. We recorded the ALB operation response times, both for MySQL and for OpenJDK, and present them in Figure 5.12.

**Increasing the balloon** involves three steps, regardless of the source application. First, the application’s balloon module acquires the application pages (e.g., InnoDB pages or Java VM eden pages). Second, the virtual addresses of these pages are passed into the kernel via a system call. Finally, the balloon driver processes each physical page and makes it available to the Xen Virtual Machine Monitor.

**Decreasing the balloon** also requires three steps. First, the virtual addresses of the reserved application pages are passed through system calls to the kernel. Second, the balloon driver maps pages for these addresses. Third, upon the completion of the system call, the balloon module in the application frees the application pages from the balloon back to the application.
## 5.3. EXPERIMENTAL EVALUATION

### Table 5.12: Latency of ALB grow/shrink operations

<table>
<thead>
<tr>
<th>InnoDB Pages (16K)</th>
<th>Grow (sec)</th>
<th>Shrink (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768 Pages (0.5 GB)</td>
<td>3.99</td>
<td>0.77</td>
</tr>
<tr>
<td>65536 Pages (1 GB)</td>
<td>7.57</td>
<td>1.51</td>
</tr>
<tr>
<td>131072 Pages (2 GB)</td>
<td>16.07</td>
<td>3.02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JVM Pages (4K)</th>
<th>Grow (sec)</th>
<th>Shrink (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>131072 Pages (0.5 GB)</td>
<td>11.28 (2.14)</td>
<td>9.48 (0.41)</td>
</tr>
<tr>
<td>262144 Pages (1 GB)</td>
<td>16.34 (4.27)</td>
<td>9.96 (0.82)</td>
</tr>
<tr>
<td>524288 Pages (2 GB)</td>
<td>18.92 (7.68)</td>
<td>10.91 (1.65)</td>
</tr>
</tbody>
</table>

For the case of InnoDB, we observe that there is a linear increase in the duration of both ALB operations with the ballooned size. The latency of the operations includes the application-level ballooning time and the kernel-level ballooning time. For InnoDB pages, the application-level ballooning time is dominated by the `memcpy` operation of the InnoDB page metadata. This cost is symmetrical for the grow and shrink operations. The large $5 \times$ difference between growing and shrinking the balloon comes from the operations that we perform in the kernel. Compared to traditional ballooning, ALB operates on pages originating from `mmap`-ed memory. Before we can place a page into the balloon, we need to remove it from the page’s zone LRU. Unfortunately, this operation can only be done under a kernel zone lock. The cost of acquiring this lock for pages ballooned out makes the grow operation more expensive and limits the achievable throughput for ballooning operations. We expect that improvements to the kernel, like the adoption of lock-free data structures [31], will reduce the time needed to perform ALB balloon-increase operations.

For the case of the Java VM, there are two numbers presented for both grow and shrink operations. The first number is the total time it takes to perform the ballooning operation (both application and kernel part). The number in parentheses gives the total time spent in the kernel for completing the ballooning operation. For the kernel time, the same remarks apply as in the case of InnoDB. The large amount of time spent in OpenJDK for completing the ballooning operation is a consequence of the full garbage collection operation that we perform before each ballooning operation in the JVM.

Figure 5.13 shows the distribution of response times for increasing, respectively decreasing the balloon in MySQL for each InnoDB page. We look at the system in idle an busy state. We ran the experiment for memory sizes of 0.5, 1 and 2 GB and recorded the times for growing and shrinking the balloon. For the “idle” case, when there was no load sent to MySQL, the latencies cluster around the smallest response times (100 to 300 nanoseconds for the balloon increase and and 10 to 30
Figure 5.13: MySQL/InnoDB ballooning operations latency histograms in busy and idle system states

nanoseconds for the balloon decrease). Under load (in the “busy” case), the same observation holds, but the tail of the distribution is fatter, with more operations taking longer. This is expected as the CPU is contended by query processing.

The throughput histograms for the balloon increase (Figure 5.14(a)) and for the balloon decrease (Figure 5.14(b)) show how the throughput is distributed while performing ballooning operations over 2GB of data (normalized to 1000 operations). The numbers were collected while JVM was handling an XMark workload and refer only to the latency of processing the kernel-side operations (end-to-end latency of the system call).

For the balloon increase operation we observe that we have two peaks in the distribution: one on the lower end (at 20MB/sec) and one on the higher end (at 170MB/sec). This bimodal distribution reaches two local maxima, indicating that the throughput of the balloon increase operation is performed in two distinct system states. We argue that this is an effect of the variable CPU usage in the case of the XMark workload.

From the balloon decrease operation we observe only one dominant peak at
1.3GB/sec which indicates that the balloon decrease operation is very fast and not influenced by the state of the pages that are being returned to the application.

5.4 Implementing ALB

This section conveys the experience in implementing the ALB modules for OpenJDK and MySQL/InnoDB as a set of engineering guidelines that can be reused in adding ALB modules to other applications.

ALB is a mechanism suited for applications that take memory management under their own control, bypassing the OS. In implementing an ALB module the developer needs to answer a set of questions:

1. What does the application use the memory for?
2. What data structure describes the free/used memory?
3. What are the policies for allocating and freeing memory?
4. How can the data structure and the policy be adapted to support memory reservation?

For most server class applications that do their own memory management we have observed that they fall into two broad categories. On one hand there are systems that use the memory for caching (databases, web or generic caching systems). On the other hand there are runtime systems in which garbage collectors use the memory for object allocation.
Among caching systems, database management systems like MySQL or Oracle rely on data buffers for faster access to data from slower storage (e.g., disks, network attached storage). Web caches like Squid 4 use main memory caches for reducing request latencies and bandwidth usage. Memcached 5 enables general purpose key-value caches for any type of applications. What all these systems have in common is that they use the main memory they have under control for decreasing request latency. The supporting data structures and policies for caching systems are often quite simple: linear lists and hashmaps with some variety of “least recently used” policy. Consequently, a memory reservation mechanism is straightforward to implement. The most complex representatives of this category are probably database systems. As we have seen with MySQL/InnoDB, the cost of implementing an ALB module was modest. For all other caching systems the approach will be similar.

Garbage collectors of runtime systems come in very many flavors: copying, mark-sweep, generational, world-stop vs. incremental vs. concurrent, etc. The supporting data structures and policies attached to them are more complex than in the case of caching systems. Still, we have shown that implementing an ALB module for a given garbage collector (i.e., PSGC) is possible with little coding overhead, as long as the semantics of the supporting data structures are understood.

5.5 Discussion

While ALB yields significant benefits, a valid concern is to what extent it generalizes to other databases or language runtime systems. A good way to frame this discussion is to distinguish between, on the one hand, the interface allowing runtime dynamic memory reallocation between database memory pools or Java VM heaps located in different virtual machines, and on the other hand, the particular implementation technique we have chosen.

In the first case, existing systems lack a specialized interface between the applications and the OS to coordinate memory use. The existing interfaces between the Operating System and applications only expose basic memory operations. The missing part is an API that allows an application to release explicitly identified memory pages, effectively giving the Operating System the information needed to make an informed choice of pages to reallocate from the database. Where the Operating System runs in a Virtual Machine, this information could be propagated

---

4Squid [111] is a web caching proxy that supports multiple web protocols that reduces bandwidth usage and request latencies by caching frequently used and accessed web content.

5memcached [79] is a general purpose, main memory, key-value based object caching system
to the Virtual Machine Monitor improving its reallocation choice. The interface needs to support the reverse direction as well, allowing the application to signal its need for more memory when required.

We argue from our results that providing such an interface has tangible benefits in resource management, and also that the interface itself can be relatively application-agnostic – there is little in the interface we have designed that is specific either to MySQL/InnoDB, OpenJDK, or to our ballooning implementation. Moreover, in Chapter 3 we show a resource management API built-in the system that allows it to interact with existing reconfiguration APIs, ranging from OS-level to VMM-level.

The second question is one of implementation, arising from the application’s assumption that memory is dedicated to its use. In databases, a mechanism complementing the existing database memory manager(s) could use an interface such as ours to dynamically adjust the size of memory pools. In Java VMs, such a mechanism could have a tighter integration with the garbage collector, up to the point where the notorious “java.lang.OutOfMemoryError” errors are handled by requesting memory from another virtual machine rather than crashing.

We sidestep such a mechanism in order to minimize the changes needed to existing code bases, and instead use application-internal routines to “allocate” memory to release via the balloon. For us, this technique has worked remarkably well, and we feel ballooning as an implementation will generalize to many (though not all) buffer managers and language runtime systems. Where it does not work, an open question is whether an alternative implementation would be possible without significant redesign.

There are certain factors that limit the amount of memory that can be dynamically shared through database ballooning as “free” pages, in fact, contain metadata. Abstracting from the storage engine used, the way memory is managed directly limits sharing:

- Increasing the page size would reduce the overhead of the page metadata. An interesting trade-off between the page size and shareable memory is worth investigating. Large storage engine pages favor sequential disk I/O (facilitating large aggregation queries or full table scans), while smaller pages speed up random I/O.

- Decoupling the metadata from the payload in the storage engine maximizes the amount of memory that can be ballooned, but can also impact performance. Touching the metadata pre-fetches the payload, since it resides on the same page. Moving the metadata into a different memory area from the
payload would require an indirect lookup for the payload page as well as a second access operation.

Also related is the question of determining the effective lower bound of memory that can remain un-ballooned in the buffer pool. More work is needed to look into these issues in order to better understand the trade-offs.

On the database side, an interesting question is what happens to query-plan optimization in the case of ALB. While the TPC-H workload we investigated did not exhibit changes in performance due a smaller buffer pool than that advertised to MySQL, there may naturally be query optimizers that could make better choices based on the current (actual) size of the buffer pool. Treating the buffer pool as a variable in the system rather than a constant might improve query optimization.

Even though the current speed of OpenJDK ALB operations is sufficient for many scenarios, we want to further investigate possible optimizations by extending the design of garbage collectors so that ALB is a first class citizen. Determining the heap generation and space in which ALB memory should be ballooned-in or -out based on specific workloads is of particular interest.

Complementing the memory managers in more complex systems with a ballooning module can be challenging. While reserving certain memory in the application is possible, the semantics for the reservation are not clear. If databases or language runtime systems are made aware of the ballooning process, some of the possible side-effects of ALB could be avoided.

A requirement (as we point out Section 5.1.2) for ALB to be useful is the correlation of memory size and performance. We assume no specific memory-vs.-performance model in this work, since we agree that this is likely application-specific, and instead view this as a policy issue. Such models are an active research area: recent papers address precisely this problem in databases ([93, 110, 33]), assuming no changes to the database. Robust behavior of databases under load is an open problem both in research and in industry [124].

An implication of this work is that if server applications can be changed to emphasize predictability and expose their cost functions (like the ideas presented in the COD system [45]), this will enable a better correlation of resource requirements with performance, and further increase the effectiveness of ALB as a mechanism for reducing over-provisioning in virtualized environments. Future systems designed for virtualized environments and architected to operate a balloon can also introduce more flexibility on both sides of the application/OS boundary.
5.6 Chapter summary

In this chapter we presented a functional miss-match between applications that manage their own memory and the consolidation capabilities offered by virtualization. We have shown that none of the existing techniques for dynamically reconfiguring the memory footprint for virtualized applications, like database engines or language runtimes, can be efficiently applied. As a solution we designed, implemented, and tested Application Level Ballooning.

We presented the challenges of dynamic memory reconfiguration in the cases of a database engine (i.e., MySQL with the InnoDB storage engine) and a language runtime systems (i.e., OpenJDK). Based on the existing memory ballooning implemented in the virtual machine monitor, ALB enables an application level balloon driver to free and reclaim memory to and from the VMM. We show that for common workloads, using Application Level Ballooning in conjunction with these systems yields the same behavior as if they were statically configured. The approach also proves that in-flight memory reconfigurations can be done between virtualized instances of MySQL and OpenJDK with very low latencies. The benefits for collocating databases are presented in a dedicated section, which is tightly related to the design on Vela.

While in the context of this thesis we use Application Level Ballooning as a technique to enable fine-grained reconfiguration in Vela, its applicability is not limited to this system.

The engineering cost and the implementation guidelines of Application Level Ballooning show that enabling other systems to take advantage of this mechanism is straight-forward and require little effort. The generality and current performance of the solution also make it a suitable candidate for many scenarios of online memory reconfiguration for service collocation in virtualized setups.
In the performance evaluation of Vela and of the Application Level Ballooning mechanism, we faced a series of performance challenges. Some of these are intrinsic to virtualization, while others are implementation details. In this chapter we focus on identifying performance overheads induced by virtualization in off-the-shelf databases and how these affect Vela.

6.1 Overview and related work

Virtualization is one of the building blocks of most modern cloud-infrastructures. Yet, few comprehensive studies investigate how it impacts the performance of commodity off-the-shelf databases. The large exploration space makes the problem even more challenging. Different virtualization solution, heterogeneous hardware, widely different database solutions, and varied workloads make exhaustive performance studies hard. Also, the fast-paced evolution of Virtual Machine Monitors and hardware support for virtualization quickly invalidate results.

Older studies have investigated the performance overhead of virtualizing databases. Minhas et al. [82] found little performance degradation due to virtualization. While they observed an overhead caused by virtualization in the case of the TPC-H benchmark and PostgreSQL, mostly due to disk I/O, in absolute values it was negligible compared to the overall query runtimes.
Che et al. [28] compare the performance of three open source VMMs: OpenVZ, Xen and KVM, chosen as representatives for three classes of virtualization: container based, para-virtualization and full virtualization. In their study they quantify the performance of the three against a non-virtualized setup using micro-benchmarking suites for main-memory, disk and network I/O as well, as application benchmarking suites like WebBench (for web servers), SysBench (for databases) or SPECC JBB (for JMs). The results show possible sources of performance loss for virtualized setups, but do not fully explore these in the context of databases.

Aboulnaga et al. [2] also make an important observations that databases and virtual machines need to be tuned together in order to be able to avoid performance degradation.

The lack of standardized benchmarks for virtualized databases has lead to industry efforts for defining ways of evaluating virtualized database performance. Recently such benchmark was defined: TPC-VMS [120]. It relies on existing TPC benchmarks (i.e., TPC-C, TPC-E, TPC-H and TPC-DS) and adds methodology for running and reporting results for virtualized databases. Our approach to evaluating virtualized database performance resembles that of TPC-VMS. Similar to TPC-VMS we rely on tree of the four supporting database benchmarks. In contrast with TPC-VMS, we also investigate the impact of different storage types. Besides full benchmarks, we also rely on micro-benchmarking to better describe certain virtualization performance aspects.

As an exhaustive evaluation is quite prohibitive, in this chapter we evaluate virtualized database aspects that affect the Vela system. As such we limit our tests to the Xen hypervisor and investigate the performance of two open source database engines: PostgreSQL and MySQL. The experimental setup also closely resembles the hardware targeted in the design of Vela: modern multicore servers. The workloads and setups that we looked at are also chosen to be representative for the different deployments of Vela.

### 6.2 Experimental setup

Our evaluation of the performance overheads of virtualized databases were carried out on a 64 core AMD Opteron having each core clocked at 2.4GHz. The machine has 512GB of main memory and 8 NUMA nodes. The NUMA node interconnect is shown in Figure 6.1. The machine has 4 sockets, with 16 cores per socket (i.e., 2 NUMA nodes per socket).

In the experiments we tested four systems: a Native system which runs a 64-bit Linux and three Virtualized systems running on top of the Xen Hypervisor (64-bit,
version 4.1.2). We used three different virtualized setups spawned on top of the Xen Hypervisor: Paravirtualized VMs (XenPV), Hardware Virtualization assisted VMs (XenHVM), and Hardware Virtualization assisted VMs with paravirtualized disk and network drivers (XenPVHVM).

In the Native setup all cores and the whole 512GB of RAM are available to the OS. In order to be able to compare the Native setup with some of the Virtualized setups, we had to limit the available resources. In a per experiment basis, we used a combination of the following tools:

- **taskset** is a tool that allows the user to specify the list of cores on which a process can be scheduled. It can be used to setup the affinity list for a new process, as well as to modify the affinity list for existing processes. For the experiments in which we are interested in limiting only the cores used by an application (e.g., allowing a database to use only 16 of the 64 cores) we use taskset.

- **numactl** is a tool similar to taskset, but offers control over both scheduling
and memory policies. In the experiments in which we want to limit the memory of an application to a specific set of NUMA nodes we use the numactl tool.

- **memhog** is a tool that we built in order to complement the existing numactl utility. numactl only allows coarse grained memory policies, at the level of NUMA nodes. There are scenarios in which we want to limit the free memory in the system. For example, when measuring disk read performance we should perform tests over datasets twice as large as the memory (in order to avoid the OS buffer cache). As our server has 512GB of RAM, running tests over 1TB datasets is impractical. The memhog utility simply callocs memory for the duration of tests, impeding the OS buffer cache to use it.

For all the virtualized setups we use the same configuration for Xen’s Domain 0. The Dom0 has a fixed amount of 4GB of memory (dom0_mem=4096M) and it has 16 CPUs (dom0_max_vcpus=16) exclusively reserved (dom0_vcpus_pin). The DomU VMs boot from a 20GB image file and have no swap. Specific memory, NUMA and CPU assignments for the VMs vary between experiments and are described accordingly.

In fixing the setup environment, we had to decide on a Linux kernel to use across all four setups (both Native and Virtualized). This proved non-trivial. First, different kernel versions yielded different behaviors. Second, many kernel configuration parameters can significantly impact the performance, favoring a Virtualized or Native setups.

We chose to evaluate two kernel versions, 3.2.1 and 3.6.6 to see how they impact the performance for a CPU and memory intensive database workload. Figure 6.2 presents these results.

Figure 6.2(a) compares the throughput of a database engine when run in four kernel setups: 3.2.1 and 3.6.6 with no paravirtualization support, and 3.2.1 and 3.6.6 with paravirtualization support. Performance wise, the paravirtualization support mainly impacts the implementation and behavior of spinlocks in the kernel. In all setups we let the OS choose the NUMA memory policy and process affinities. We observe that the performance of the 3.2.1 kernel has a performance spike at 8 clients for the setup that does not offer paravirtualization support.

Figure 6.2(b) is similar to the previously described experiments, but we actively bound the database processes to certain cores. By keeping the database processes on the same NUMA nodes we see a slight performance increase for all setups. Under the investigated workload the database uses many locks concurrently. When the memory of these locks is touched from the same NUMA node, the latency is reduced as compared to remote accesses.
Figure 6.2: Kernel versions, hard NUMA affinity and kernel support for paravirtualized spinlocks

Figure 6.2(c) groups together the four setups of the kernel version 3.2.1, while Figure 6.2(d) does the same for the kernel version 3.6.6. On one hand we observe that in the 3.2.1 kernel version the database performance is greatly impacted by the manual NUMA policies and by the effect of paravirtualization support. On the other hand in the case of the 3.6.6 kernel version the database performance is more stable, though lower for low degrees of concurrency. Based on these findings, we chose to use Linux kernel 3.6.6 in all our experiments as it yielded more stable performance. For the virtual machines, we used the kernel compiled with paravirtualization support for synchronization, while the non-virtualized setups use a kernel without this flag enabled (but otherwise identical).
6.3 Micro benchmarks

Before diving into an actual suite of database benchmarks we performed a series of micro-benchmarks. The purpose of the micro-benchmarks is to understand the performance of individual system components in both virtualized and non-virtualized setups. We focus this experiments on CPU, memory, disk, and network performance.

For understanding the source of overhead when running databases in virtualized environments we analyze separately the performance of different storage types, memory access and networking.

For durable storage, we look at the performance of Network Attached Storage mounted over NFS and local Solid State Drives (directly mounted in the Virtual Machines in the Virtualized setups). For the virtualized setups we also investigate the performance of Virtual Disk Images stored on a local Solid State Drive mounted in Xen’s management domain Dom0. For the durable storage evaluation we investigate sequential read and write performance of the four systems (one Native and three Virtualized).

For the memory access performance study we compare three of the four described systems: the Native setup and two Virtualized setups (XenPV and XenPVHVM). Similarly for the network performance we compare the same three systems as in the case of memory: Native, XenPV and XenPVHVM over 1Gbit and 10Gbit networks.

6.3.1 I/O write performance

The durable write performance is useful in understanding the I/O subsystem of database engines. In the design of Vela we mention that the I/O subsystem is a possible bottleneck for the transaction logs of the Master database. As the load on the transaction log is a sequential in nature, we present the performance difference of sequential writes between the four studied systems over three different storage mediums.

For this evaluation we used the Bonnie++ [14] tool. Bonnie++ is a benchmarking suite for performing a series of tests for evaluating the performance of disk drives and file systems. For all the experiments the options used for Bonnie++ are described in the table in Figure 6.3.

For the three Virtualized setups, we used VMs configured with 4 cores (on the same NUMA node) and with 8GB of RAM. As in this experiment we are focusing on I/O subsystems, memory plays little role. In the Native case, the Bonnie++
6.3. MICRO BENCHMARKS

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r</td>
<td>0</td>
<td>Do not check actual system RAM size. We use this option as we manually take care of correlating the test data size with the RAM available on the system.</td>
</tr>
<tr>
<td>-c</td>
<td>1,16</td>
<td>The concurrency level. This defines how many concurrent threads Bonnie++ will spawn for running the same workload.</td>
</tr>
<tr>
<td>-d</td>
<td>/mnt/NFS /mnt/SSD /mnt/VDI</td>
<td>The root mount point for the experiment. This points Bonnie++ to the directory that is used for storing the files used for benchmarking. We benchmark NFS, SSD and VDI backed storage.</td>
</tr>
<tr>
<td>-s</td>
<td>16GB</td>
<td>The size of the data used in the test. This needs to be correlated with the available RAM in the machine, such that for read tests the file is not completely stored in RAM. It is less important for the write tests.</td>
</tr>
<tr>
<td>-b</td>
<td></td>
<td>Flag indicating no write-buffering, forcing an fsync() after each write operation. This is also how databases generally write to the transaction log for ensuring durability</td>
</tr>
<tr>
<td>-D</td>
<td></td>
<td>Flag indicating that Bonnie++ should use DIRECT IO for opening the files. With DIRECT IO there is no caching of the data files used in benchmarking. This is the most common way for databases to open the files they access in order to avoid double caching (at the database level and in the OS buffer cache)</td>
</tr>
</tbody>
</table>

**Figure 6.3:** Bonnie++ configuration options

The process was bound to 4 cores using the taskset tool while the free memory was also reduced to 8GB using our memhog tool.

Figure 6.4 presents the performance of the four studied systems over the three different storage types. Figure 6.4(a) shows the sequential write throughput with 1 concurrent Bonnie++ thread running while Figure 6.4(b) shows the throughput for 16 concurrent Bonnie++ threads.

First, we remark that the behavior of the four systems over the three storage types does not change by moving from 1 to 16 concurrent threads.

Second, we clearly observe that the performance of XenHVM is way below that of the other systems. Given this, we will only focus the rest of the experiments on XenPV and XenPVHVM as virtualized setups. The presence of the paravirtualized
Xen drivers for disk and networking greatly boost the performance of sequential writes for XenPV and XenPVHVM – which behave almost identically. The performance loss between the Native and XenPV and XenPVHVM is relatively small for the NFS storage, though quite large for local SSD storage.
Third, the Virtual Disk Image performance is similar to that of the local SSD Disk, which makes sense since the VDI file is stored on an identical SSD and the one used in the SSD Disk setup. Differences between the two would be only noticeable with multiple Virtual Machines. While in the case of VDIs, multiple VMs would contend for the backing disk, the case of locally mounted SSD disk makes it exclusively available to one VM.

The conclusion of this micro-benchmark is clear: for small, fsync-ed sequential writes, similar to those performed by the transaction logs of databases, the Native setup has a throughput that is almost 2 times higher than that of the best Virtualized setup (Native at $\approx 154$MB/sec vs. Virtualized at $\approx 88$MB/sec) for local SSD disks. In the case of NFS storage, the throughput of the Native setup is 1.3 times higher than that of the best Virtualized setup (Native at $\approx 32$MB/sec vs. Virtualized at $\approx 25$MB/sec). It is a clear indication that we should expect a lower performance for virtualized databases running update intensive workloads bound on the transaction log, as compared to the non-virtualized case.

### 6.3.2 I/O read performance

The read performance is of less importance for Vela. By design our system tries to cache most of the data in main-memory. Still, caching the data initially in RAM requires reading it from durable storage. To this extent we present a performance comparison of the Native and Virtualized (XenPV and XenPVHVM) setups over local SSD driver and VDI images.

The system configurations as well as the Bonnie++ settings are the same as for the sequential writes evaluation. The only difference is that for the read performance evaluation we need to ensure that we actually go to the actual storage medium for reading. To ensure this, the size of the processed data set needs to be larger than the available RAM. This is ensured as the test data is 16GB and all setups only make less than 8GB available to the Bonnie++ process.

Figure 6.5 presents the comparison of the sequential read performance (throughput in MB/second) benchmarked with Bonnie++ for the three studied systems. We repeated the evaluation for 1 and 16 concurrent Bonnie++ threads.

The Native system performs best, though the XenPVHVM performance is just slightly worse, with XenPV being the slowest, both for no-contention (1 Bonnie++ process) and higher-contention (16 Bonnie++ processes) scenarios. In the case of VDI images the sequential read throughput peaks at 100MB/second, with both XenPV and XenPVHVM performing identically, with a small gain for XenPVHVM (as in the case of the SSD reads).
6.3.3 Memory performance

In evaluating the raw performance of accessing main memory from Native and Virtualized environments, we explored different possible benchmarks, either in user space or in kernel space. We opted for a kernel-space evaluation since it is
prone to the least amount of noise. Consequently we implemented a Linux kernel module that measures the time required to initialized an array with a constant value, as described in the pseudo-code in Figure 6.6.

**KERNELMEMTest**(chunkSize, totalSize)

1. `src ← kalloc(chunkSize)`
2. `dest ← kalloc(totalSize)`
3. `count ← totalSize/chunkSize`
4. `start ← GETCYCLES`  
   
5. `for k = 0; k < count; ++k`
6. `do`
   
7. `memcpy(src, dest + k * chunkSize, chunkSize)`
8. `return(GETCYCLES − start)`

**Figure 6.6:** Kernel memory access benchmarking pseudo-code

Figure 6.7 shows the time for completing the memory initialization for a “total-Size” of 1.5 GB while varying the “chunkSize” from 4 bytes to 16 KB. We evaluated the performance of a Native system versus that of two Virtualized systems (XenPV and XenPVHVM) on two hardware platforms. One is the previously described 64 core AMD Opteron (Figure 6.7(a)) and the other is an 8 core Intel (Figure 6.7(b)).

We are not comparing the two server architectures, so we do not care about the difference between the absolute values for the Intel and AMD servers. In the case of AMD, we notice that the performance of the two Virtualized setups (XenPV and XenPVHVM) completely overlap over the whole range of Chunk Sizes. The Native setup behaves differently in the range of 4 to 32 byte chunks. For small (4 byte) chunks it is slower than the virtualized setups. From 8 bytes onward it is always faster (even if marginally as the chunk sizes increase).

Intrigued by the repeatable slower behavior of the Native setup for 4 byte chunks we wanted to see if this was an artifact of the server we used or something inherent to the Linux kernel or Xen. By repeating the same experiment on a different hardware architecture (on the Intel server) we did not observe this behavior, noting that the Native setup is always the faster of the three.

We draw two conclusions from this micro-benchmark. First, we observe that performance differences between the studied systems is highly dependent on the hardware architecture. Though we see a weird performance artifact for operating on small chunks, the performance difference is small enough to be negligible. This leads us to the second conclusion and hypothesis: main memory database workloads should behave almost identically between Native and Virtualized setups. We explore this hypothesis in Section 6.4.1.
6.3.4 Network performance

For evaluating the performance of Virtualized setups vs. Native setup in terms of networking performance we used the netperf and netserver tools [87]. Netperf is a benchmarking tool that can be used to determine the performance characteristics of different types of networks. Of particular interest in the scope of the Vela system are TCP/IP networks. The results presented in this section refer to TCP over IPv4.
6.3. MICRO BENCHMARKS

in 1Gbit and 10Gbit networks.

All experiments consist of the netperf clients connecting and sending out packets of different sizes to a netserver, residing on a different machine. We ran concurrently multiple netperf clients (varying between 1 and 16) that send out packets of sizes varying between 16 bytes and 2 Kbytes. These packet sizes are chosen as they represent different message sizes in the range of messages exchanged in Vela. We repeated the tests with 1Gbit and 10Gbit links between the netperf clients and the netserver.

The measurements are performed from the client's perspective, which runs the netperf tool. In our evaluation we ran the netperf tool from the Native, XenPV and XenPVHVM setups.

Figure 6.8 presents the performance (measured as the achieved throughput in Mbits/second) for 1Gbit (left column figures) and 10Gbit links (right column figures), for 1, 8, and 16 concurrent netperf clients. Each of these six experiments compares the achieved throughput of the three investigates systems while varying the packet size.

First we analyze the 1Gbit link experiments (shown in Figures 6.8(a), 6.8(c), and 6.8(e)). For all three experiments we remark that the XenPVHVM setup closely resembles the behavior of the Native system, reacting in a similar way to changes in both netperf client count and packet size. In the case of 1 client and very small packets (16 bytes), both the Native and the XenPVHVM setups are limited by the CPU of the core running the netperf client. In the case of XenPVHVM we observed both the VM CPU running the netperf client utilized at 100% (mostly in `sys`) as well as the VMM CPU handling the Xen netback process (mostly in `irq`). Actually using two CPUs (one in the VM and one in the VMM) instead of one, XenPVHVM slightly outperforms the Native setup. The Native setups are CPU bound only for the 1 netperf client with 16 byte messages. In all other cases the throughput is bound by the network bandwidth. For XenPVHVM, for messages larger than 16 bytes the throughput is also bound by the network bandwidth. For the 16 byte messages, there is a shifting bottleneck from the CPU usage of the netback in Xen's Dom0 to the CPU `irq` in the VM – as the number of netperf clients increases. In contrast with Native and XenPVHVM, the XenPV setups are more CPU demanding. For all experiments with 16 byte messages and for the 128 byte message with 1 netperf client, the bottleneck is both on the CPU in the VM and on the netback CPU in Dom0. All the other setups become bounded by the network bandwidth.

For the 10Gbit network link experiments (shown in Figures 6.8(b), 6.8(d), and 6.8(f)) we see that the XenPVHVM no longer follows the Native performance as closely. It resembles more the performance of XenPV (though performing better for
Figure 6.8: Network performance varying concurrency, packet size and network bandwidth: Native vs. XenPV vs. XenPVHVM

lower packet sizes). As we removed the network bandwidth bottleneck from many of the previous experiments, the setups face new bottlenecks. The Native setup is
bottlenecked on the netperf client CPUs for all 16 byte message size experiments. For the 128 byte message size the Native setup is also bottlenecked by the netperf client CPU sys for 1 thread and on CPU irq for 16 threads. As the number of clients increases, we see a shift – in the case of small messages (16 and 128 bytes) – from CPU sys to irq time. For all other Native setups we reach the network bandwidth limit. Both Virtualized setups are shifting the bottlenecks between the CPU usage of the netback in Dom0 and the CPU usage of the netperf clients in the VM. In the case of XenPV, the throughput is always limited by the netback process utilizing 100% a CPU – shifting from irq time for small packet sizes to sys as the packet size increases. XenPVHVM behaves similarly, but shifts faster from irq to sys dominating the CPU usage for the netback, as well as exhibiting lower total CPU usage by the netperf clients in the VM. These explain the constantly higher throughput achieved by XenPVHVM as compared to XenPV.

We draw a series of conclusions from these sets of experiments. First, for 1Gbit networks, the performance overhead of virtualization is negligible, with 1 netback process in the Dom0 being sufficient to drive a virtual network interface in the VM to fully utilize the network bandwidth, assuming messages larger than 128 bytes. For lower messages we have seen degraded performance even in the Native case. Second, for 10Gbit networks, virtualization takes its performance toll. The bottleneck is the single netback process per virtual network interface in a VM. In the next section we investigate a network-bonding based solution that alleviates the problem for certain scenarios.

6.3.5 Bonding virtual network interfaces

In Xen DomUs (i.e., VMs) an arbitrary number of virtual network interfaces (VIFs) can be defined. Each VIF will be seen by the OS in the VM as a different network interface. Also each VIF will be handled in the control domain 0 (Dom0) by exactly one Xen network back-end driver (“netback”) running on a kernel-thread. The total number of netback threads is equal to the total cores available in Dom0. Based on this design of Xen we try to overcome the CPU bottlenecks that we presented in the previous section in both Dom0 and DomUs.

First, we want to balance the CPU cost in Dom0 of a guest over multiple netback threads. We showed that for a 10Gbit network, the limiting factor for the XenPVHVM performance was the netback CPU cost in Dom0. For this we rely on network bonding over multiple VIFs assigned to the guest DomU. We modify the XenPVHVM’s DomU configuration such that the guest OS will see 2, 4, or 8 network interfaces. In the guest OS we use the Linux “bonding” network driver to create a “bond” interface that uses a round-robin policy to send out packets over
Figure 6.9: Effects of multiple VIFs and network bonding in 10Gbit network with XenPVHVM
all the available network interfaces. Our hypothesis is that this will distribute the
CPU load in Dom0 over multiple netback threads, which in turn will allow the
guest to achieve higher throughputs.

Second, we want to better balance the irq CPU costs in the guest over all
available cores. For achieving this, we applied most of the optimizations described
in the Xen network performance guidelines [131]. We also used the irqbalance
d daemon for interrupt balancing and manually set the “smp_affinity” of the “xen-
dyn-events” for each virtual network interface. The SMP affinities of the Receive
Packet Steering and Transmit Packet Steering SMP were also distributed over all
available cores. A final set of optimizations are directly related to the round-robin
policy used in conjunction with the network interface bonding. As suggested in
the documentation of the bonding driver, we investigated different settings for the
TCP packet reordering (“net.ipv4.tcp_reordering”).

In the experiment we used a Dom0 with 16 cores (implicitly supporting 16 net-
back kernel threads) and 4GB of RAM. The network performance was measured
using the netperf tool from within a VM running XenPVHVM with 16 cores. We
varied, like in the previous experiments, the number of concurrent netperf pro-
cesses, as well as the size of the processed packets. Figure 6.9 presents our findings
for 1, 8, and 16 netperf clients for packets of 16, 128, 1024, and 2048 bytes. We
compare 4 setups, with 1, 2, 4, and 8 VIFs. Figure 6.9 shows the throughput that
was achieved in all of these configurations.

The results confirm our hypothesis only for extreme cases. For the case of low
contention, when only 1 netperf client is issuing load, the increased number of VIFs
yields better performance. For small packet sizes (16 and 128 bytes) extra VIFs
improve the maximum achievable throughput, while for large packet sizes 8 VIFs
yield the best performance. In the case of heavy contention, with 16 netperf clients,
increasing the number of VIFs improves overall performance for small packets of
16 bytes. For each VIF setup, we observe that given enough load it will saturate
and stay constant in throughput once a given packet size is reached.

What we did not formulate in our initial hypothesis is the overhead of bonding.
The added latency of the bonding driver and its round-robin policy negatively
impact performance in the VMs. While CPU costs are better balanced over net-
back threads in Dom0, the added latency of bonding ends up hurting performance
in many cases. Still, for the case of heavy concurrency and small message sizes,
bonding does help. This is exactly the scenario for which we used bonding for
enhancing the performance of Vela.

While bonding is a solution for further pushing networking performance in VMs
at the cost of extra CPU utilization in Dom0, it is not applicable for all workloads
and requires a large amount of engineering and tuning. We consider that improve-
ments in both hardware [75] and software [84] support for network virtualization will allow VMs to take full advantage of 10Gbit network infrastructures.

The approach of bonding virtual network interfaces is derived from limitations that we observed in the Xen hypervisor. While using different hypervisors might not lead to similar approaches, in the case where there is no hardware support for virtualized network I/O, similar issues with regard to interrupt balancing and CPU usage (in both host and guest) need to be considered.

6.4 Virtualized database performance

Having presented the performance of individual system components in virtualized and non-virtualized environments, we investigate how actual database systems perform under a variety of workloads.

In this section we present the performance two very popular open-source off-the-shelf database engines: PostgreSQL (version 9.2) and MySQL (version 5.5), both of which are supported by Vela. We investigate three storage setups for both database engines: main memory, local SSD disk, and network attached storage. For each of these scenarios, we choose database workloads that stress the storage, and show how the databases' performance is influenced by running the engine virtualized compared to non-virtualized.

6.4.1 Main memory storage

Based on the hypothesis that databases operating over data held entirely in main memory should not see any performance degradation due to virtualization, we explore the behavior of PostgreSQL and MySQL with an OLAP workload (TPC-H [119]) and a transactional e-commerce inspired workload (TPC-W [121]). These set of experiments give insight on the performance of the Satellites, Observer and HotBackup DPIs of Vela. As these replicas do not contribute to Vela’s durability, they are always held in main-memory.

OLAP workload

OLAP workloads are characterized by heavy long running queries that operate over large amounts of data. Such workloads stress both the CPU and the data access paths. We chose TPC-H, a standard database benchmark to compare the OLAP performance of PostgreSQL and MySQL in two virtualized setups (XenPV and XenPVHVM) to their Native (non-virtualized) performance.
Figures 6.10(a) and 6.10(b) show the individual response times of each of the TPC-H defined queries for PostgreSQL and respectively MySQL.\footnote{In the case of MySQL we were unable to run all the 22 queries specified by the TPC-H benchmark. Queries 5, 7, 9, 12, and 18 are omitted since their runtimes were prohibitively long. However, we consider that this omission does not weaken the results.} The reported
values are averaged over 5 runs, over a dataset of approximatively 16GB (5 GB of raw data) corresponding to a TPC-H scale factor of 5. For all experiments, the load was generated by 32 concurrent clients. The datasets were entirely stored in main memory in tmpfs. For the virtualized setups, the VMs were allotted 2 entire NUMA nodes: 16 cores and the corresponding 64 GB of RAM. For the Native case, the database engines were pinned to the same 2 NUMA nodes (both CPU and memory wise).

For both PostgreSQL and MySQL we observe that the response times of each query are very similar for all three setups. The error bars and the noise in the plots are a consequence of the randomness in the query parameters. For the same query, different parameter values can lead to widely different run-times and can also generate different execution plans. Also, the databases are always processing 32 queries at a time, leading to contention on the 16 CPUs. This also generates different inter-leavings of the queries. Nonetheless, we interpret the results for both MySQL and PostgreSQL as being very similar in all three setups.

E-Commerce workload

The previously investigated OLAP workload stressed memory access and long running queries that fully utilize the CPU. For fast, transactional workloads that stress both reading and writing to main memory, we rely on an e-commerce inspired workload. TPC-W with its different mixes of transactions enables us to do this evaluation. As in the previous experiment, we compare the same databases in the same three setups.

Figure 6.11 shows the average throughput and response times for PostgreSQL and MySQL under the three setups while varying the number of clients issuing a TPC-W Browsing mix load. The Browsing mix is a read-mostly workload (with only 5% updates). This workload mix emphasizes that the CPU and memory read performance is not affected by virtualization, allowing both PostgreSQL and MySQL to achieve the same performance as in the non-virtualized setup.

Figure 6.12 shows the average throughput and response times for PostgreSQL and MySQL under the three setups while varying the number of clients issuing a TPC-W Ordering mix load. Compared to the previous workload mix, the Ordering mix also stresses the memory write performance. Its transactions generate many changes in the data as well as many appends to the transaction log (which in this experiments is also main-memory resident).

The results confirm our hypothesis: virtualized databases do not incur a performance penalty for operating entirely out of main memory, compared to the non-virtualized case. For Vela this means that the Satellites, Observer and Hot-Backups suffer no performance penalty due to virtualization.
6.4. VIRTUALIZED DATABASE PERFORMANCE

![Graphs showing performance metrics for PostgreSQL and MySQL]

(a) PostgreSQL – Throughput
(b) MySQL – Throughput
(c) PostgreSQL – Response Time
(d) MySQL – Response Time

**Figure 6.11:** Performance of PostgreSQL and MySQL for main memory storage running a read-intensive e-commerce workload (TPC-W Browsing mix): Native vs. XenPV vs. XenPVHVM

### 6.4.2 Local disk storage

Based on the observations from evaluating the local disk read and write performance of Virtualized vs. Native setups, we hypothesize that under heavy I/O workloads, the performance of virtualized database engines durably storing the transaction logs on a local disk will have a degraded performance as compared to non-virtualized setups. In this section we prove this hypothesis by running a workload based on the TPC-C [117] Benchmark against PostgreSQL and MySQL.

The TPC-C workload comprises five fast transactions that generate pressure on the database’s transaction log. The transaction logs of both PostgreSQL and MySQL are stored on local SSD storage. In the Virtualized setups the SSD disk is mounted directly and exclusively in the VM running the database. The dataset used in the experiments corresponds to a TPC-C scaling factor of 100 warehouses.
Figure 6.12: Performance of PostgreSQL and MySQL for main memory storage running an update-heavy e-commerce workload (TPC-W Ordering mix): Native vs. XenPV vs. XenPVHVM

(which represents 10GB of raw data, not accounting for extra space taken up by indexes).

The experiment consists of clients issuing load according to the TPC-C workload and measuring the response times and throughput of the two databases in three setups: Native, XenPV and XenPVHVM.

Figure 6.13 presents the experimental data. We clearly observe that for the whole range of clients, the throughput and response times of both PostgreSQL and MySQL are the best in the Native case. The gain is smaller in the case of PostgreSQL and more substantial in the case of MySQL. As the databases durably write to the transaction log on the SSD after each completed transaction, the latency of this operation determines the throughput (and the response times).
Figure 6.13: Performance of PostgreSQL and MySQL for local disk storage running a write-intensive OLTP workload (TPC-C): Native vs. XenPV vs. XenPVHVM

These results confirm our hypothesis and corroborate our findings regarding the
sequential write throughput in Native vs. Virtualized setups. These results directly impact the performance of Vela’s Master DPIs, which need to offer durability in the system.

6.4.3 Network storage

The results from the network performance micro benchmarks indicate that there is a potential performance overhead for virtualized databases when accessing data from network storage. This overhead is mostly noticeable in the case of 10Gbit than in 1Gbit networks. We hypothesize that both XenPV and XenPVHVM will show a degraded performance as compared to Native when the databases are handling load for datasets stored on network attached storage.

To investigate the hypothesis, we use PostgreSQL and MySQL, processing a TPC-H [119] workload \(^2\) over a scale-factor 5 database (i.e., 5GB raw data and approximatively 16GB dataset size, including indexes). The whole dataset is stored on a network attached storage, accessed through NFS. In all setups, both databases were configured to cache in main memory 4GB of data. The total memory in the VMs was 8GB. Using the “memhog” tool we also limited the available memory in the Native setup to 8GB. Two concurrent clients were issuing the TPC-H workload to the databases.

Figure 6.14 shows the response time, for each query, under the three different setups. The bars represent the average value observed for each query, along with the standard deviation error-bars. The y-axis of the plot is logarithmic as the response times vary by orders of magnitude between different queries. Compared to the experiments where the data was stored in main-memory (Figure 6.10), we first observe the higher error-bars. This is a common artifact in disk based storage systems (as is the case of the network attached storage used in this experiment) which have variable access latencies. Even though in some cases the error-bars overlap, the trend is clear for both database engines: the Native has a significant performance advantage compared to the Virtualized setups. The average response time over all the queries was 60% (for MySQL) and 30% (for PostgreSQL) higher in the Virtualized setups compared to the Native setup.

Figure 6.14(a) shows how PostgreSQL performs in the three tested setups. Besides the general remark that Native is faster then Virtualized setups, we also observe a consistent pattern of XenPVHVM being slightly faster than XenPV. This corroborates our micro-benchmark observations on network performance. Figure 6.14(b) shows how MySQL performs in the same three setups. In the case

---

\(^2\)As in the other experiments with MySQL processing a TPC-H workload, we did not run queries 5, 7, 9, 12, and 18 as their run-times are prohibitively large.
Figure 6.14: Performance of PostgreSQL and MySQL for network attached storage running an OLAP workload (TPC-H): Native vs. XenPV vs. XenPVHVM

of MySQL we observe the opposite: XenPV tends to perform better than XenPVHVM. We speculate that the Asynchronous I/O library (i.e., “libaio”), used by MySQL and the InnoDB storage engine for accessing the data files, favors XenPV over XenPVHVM. In a simple experiment we configured MySQL/InnoDB with a 1GB buffer pool for a 16GB dataset, forcing the database to go to storage
(main-memory) for almost all the pages. In this experiment we are stressing the “libaio” library and minimizing storage latency interference by storing the data in main-memory. The result was a very consistent $1.2 \times$ higher response times for XenPVHVM compared to XenPV.

Based on the above experimental results we conclude that the hypothesis was proved: virtualized databases incur a performance penalty in the case of datasets accessed over a TCP/IP network. Also, the experiments point out a divergence in the behavior of virtualized PostgreSQL and MySQL, which we speculate is rooted in the “libaio” library.

6.5 Chapter summary

In this chapter we analyzed the performance impact of virtualizing database engines. As an exhaustive evaluation is far too costly from a time perspective, we focused on the database systems and virtualization solutions we used in Vela. Our results are grouped into three main categories: system configuration, micro-benchmarks, and virtualized database performance.

With regard to system configuration, we have shown that different Linux kernel versions and kernel configuration parameters lead to widely different system performance results. We consider that as the kernel offers more virtualization support, the impact of kernel versions on application performance will be diminished.

The micro-benchmarks that we conducted gave us valuable insight in understanding and overcoming performance bottlenecks. We confirm previous results that the I/O read and write performance in the case of virtualization is lower than for native setups. We also show minimal performance difference for main-memory access, though curiously we discovered hardware specific performance artifacts. From the perspective of networking performance we found that Xen based virtualization solutions are optimized for 1Gbit networks. For 10Gbit networks, the virtualized setups hit performance bottlenecks. With the current design of virtualized networks these bottlenecks can be worked-around. Our efforts in doing so yielded higher throughputs for corner cases, which matched common-cases of Vela. Yet, most of the times the engineering of removing the virtualization bottlenecks only adds more latency, effectively reducing or even canceling its benefits.

The micro-benchmarks enabled us to formulate multiple hypothesis regarding the performance of virtualized databases, under various workloads and deployed on different storage types. We empirically showed that when using main-memory storage for both dataset and transaction logs, virtualized databases perform similarly to the non-virtualized case. We proved this for OLAP workloads, as well as for
e-commerce workloads varying from read-intensive to update-heavy. In the case of durable setups, where the database’s transaction log is held on local SSD storage, while serving an OLTP workload, virtualized setups exhibited a lower throughput and higher transaction latency. As hinted by the I/O disk write micro-benchmarks, this is caused by the sequential I/O overhead of synchronously writing the transaction log entries to disk. Finally, for large datasets accessed from a network attached storage, the network itself proved to be an overhead. As proved by the results from an OLAP workload, under which the database constantly accesses the storage over the network, virtualized databases show an increased latency in answering queries. The empirical results of databases, relying on different libraries for accessing the datasets, indicate that different types of virtualization play a role in the databases’ performance.
Conclusion

In this thesis we have addressed a series of problems related to data processing in cloud infrastructures. While most cloud-ready solutions promoted in the form of NoSQL databases are very well aligned to cloud computing characteristics they bear their own set of limitations in the form of crippled functionality, limited APIs or reduced consistency guarantees. However, these limitations are strong points of traditional off-the-shelf relational database management systems. In many cases it is highly desired to be able to run off-the-shelf database engines in cloud environments to take advantage of cloud computing scalability, elasticity or support for multi-tenancy.

To take full advantage of cloud infrastructures, traditional database engines need to align themselves to the characteristics of cloud computing. Unfortunately, naively deploying off-the-shelf database solutions in cloud infrastructures does not result in either an alignment to cloud characteristics, and even worse in many cases it results in performance degradation and functional miss-matches.

Motivated by these discrepancies between cloud-ready NoSQL databases and traditional relational database management systems, we designed, implemented and evaluated a solution that offers the functionality of off-the-shelf databases, while also being aligned to cloud computing requirements. The novel approaches of our solution are presented in the Vela system (and in Multimed, its predecessor) as well as in the Application Level Ballooning mechanism.

Vela is a highly scalable system that embraces both the number of servers and
their size (with respect to cores and memory) as scaling infrastructure. Based on a primary-master asynchronous snapshot isolation data replication model, our system scales out over servers as well as up with the computational resources of individual multicore machines.

Relying on replication rather than parallelization, Vela addresses the problem of efficiently running databases on multicore machines. The system represents a departure from existing work in that it solves the problem for a wide range of loads without having to modify the engine. It uses off-the-shelf databases in a replicated configuration and deploys them over a multicore machine as if the multicore machine were a distributed system. As shown in the evaluation, it exhibits better and more stable performance on multicore architectures than PostgreSQL and MySQL.

Based on the chosen replication model, we showed that Vela seamlessly scales out over clusters of both commodity and high-end multicore servers, improving the system’s performance by balancing the load over all the available resources and supporting multi-tenancy. By supporting multiple replicas running on the same physical machine, we show Vela takes advantage of running off-the-shelf databases at their optimal size (with respect to cores and memory). We have shown that this allows it to overcome issues in synchronization contention and load interaction.

Besides being able to scale, we have shown that Vela embraces virtualization both in its deployment and in its reconfiguration ability. Using virtualization the system’s components encapsulated in VMs run in isolation and all available hardware is modeled as a uniform pool of resources. Through Vela’s reconfiguration API, based on existing as well as novel virtualization mechanisms, this pool of resources can be re-allocated among the system components in runtime at different levels of granularity. In the evaluation of Vela we exemplified how the reconfiguration API can be manually or automatically triggered to re-shape the system such that it can self-scale to meet service level agreements. In the thesis we investigated the direct impact of virtualization on databases from the perspectives of performance and functionality. From the performance perspective we showed how Vela’s design and implementation overcomes sources of performance degradation.

From a functional perspective we showed that fine grained memory management mechanisms supported by virtualization are ill-suited for off-the-shelf databases and language runtimes. As Vela relies on both and emphasizes the benefits of online memory reconfiguration, we developed a new virtualization based mechanism, Application Level Ballooning, to efficiently vary the memory available to two classes of server application: databases (MySQL) and language runtime (OpenJDK). We showed that the resulting performance is no different to that of a statically configured application of the same size, with no need for restart with a new
configuration. ALB piggy-backs onto existing OS-level ballooning, creating an end-to-end solution for memory reallocation, coordinated across all three software layers: VMM, OS, and application. The changes required to implement ballooning are small and limited to the OS (and its balloon driver) and application itself. Our experiments show that ALB dynamically moves memory between applications to scale performance between low and peak loads. The adaptation speed is determined by the workload and the ability of the application to rapidly exploit recently-freed memory. We show ALB is agile enough to move memory between two tiers of an application to optimize end-to-end performance.

In the case of database collocation, we showed that ALB can be used to react to changes in the workload such that system throughput does not degrade. This is achieved by on-the-fly buffer pool resizing rather than having an over-provisioned buffer pool, facilitating fine grained memory reconfiguration options for Vela’s Data Processing Instances.

The system evaluation showed that Vela handles diverse read-intensive workloads and exhibits little overhead from virtualization. Paying attention to engineering aspects, both in implementing and deploying the system, we were able to achieve very good scalability and high throughput rates while supporting transactional workloads with snapshot isolation guarantees.

A key aspect of Vela is that it is independent of the database engine and it will benefit from current hardware developments, something that is not always the case for alternative approaches. Vela gets better as the number of cores increases, as the number of cluster nodes it controls increases, implicitly as more main memory is available, through network attached storage, and by using SSD/Flash storage. In addition, it is in a better position to cope with the impending heterogeneity of both multicore machines by allowing asymmetric replicas of the database that can be specialized to the characteristics of the underlying cores, or of the cluster through virtualization.

In the interest of building a functional system, we left many interesting research avenues open. These include workload optimizations through Satellite specializations at the storage layer (row-storage vs. column-storage), or through support for different index types, query optimizations and query operators (time-travel, sky-line, or full-text search). The applicability of specialized Satellites is not limited to performance and functional aspects. Vela could also be used for Byzantine fault tolerance or extended to support geo-replication. From the database virtualization perspective, there are opportunities to investigate ways to mitigate performance aspects (most notable that of virtualized network I/O).

We hope that these ideas will inspire future researchers in the field to take the system even further!
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction: Context overview of the presented solution</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Motivation: Virtualized database performance overhead for non-durable main-memory resident datasets</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Motivation: Virtualized database performance overhead for durable local SSD resident datasets</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Motivation: PostgreSQL’s scalability on modern multicore servers – effects of load interaction</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Motivation: Database scalability on modern multicore servers – effects of synchronization contention</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Motivation: The effect of conventional memory ballooning on database query execution performance</td>
<td>29</td>
</tr>
<tr>
<td>2.6</td>
<td>Motivation: Effect of JVM heap size on XMark response time</td>
<td>30</td>
</tr>
<tr>
<td>2.7</td>
<td>Motivation: Effect of DBMS memory size on TPC-H query response time</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Vela design: replication systems feature-set</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Vela design: Overview of the replication mechanism</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Vela design: Master commit and WriteSet propagation algorithm</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Vela design: Router load balancing algorithm</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Vela design: System Model XML snippet</td>
<td>47</td>
</tr>
<tr>
<td>3.6</td>
<td>Vela design: Automatic Satellite core count scaling algorithm</td>
<td>55</td>
</tr>
<tr>
<td>3.7</td>
<td>Vela deployment example: core partitioning in a multicore machine</td>
<td>56</td>
</tr>
<tr>
<td>3.8</td>
<td>Vela deployment example: using VMs for partitioning a multicore machine</td>
<td>57</td>
</tr>
</tbody>
</table>
3.9 Vela deployment example: VMs over a cluster of multicores 58

4.1 Vela evaluation: PostgreSQL standalone vs. Vela, running the TPC-W Browsing mix 66
4.2 Vela evaluation: PostgreSQL standalone vs. Vela, running the TPC-W Shopping mix 68
4.3 Vela evaluation: MySQL standalone vs. Vela, running the TPC-W Browsing mix 70
4.4 Vela evaluation: MySQL standalone vs. Vela, running the TPC-W Shopping mix 71
4.5 Vela evaluation: Vela overhead, TPC-W Browsing mix 74
4.6 Vela evaluation: Effect of load separation in Vela, running TPC-W Browsing mix 75
4.7 Vela evaluation: Scale up performance of PostgreSQL v8.3 vs. v9.3, running a read-intensive workload 76
4.8 Vela evaluation: Scaling up with number of cores, PostgreSQL standalone vs. Vela, running TPC-W Browsing mix 78
4.9 Vela evaluation: Scaling up with number of cores, PostgreSQL standalone vs. Vela, running TPC-E 79
4.10 Vela evaluation: Effect of overloaded Satellites 80
4.11 Vela evaluation: Monitoring the Master load as we scale-up 80
4.12 Vela evaluation: Scaling out over a cluster of multicores, Vela running TPC-W Browsing mix 82
4.13 Vela evaluation: Monitoring the Master load as we increase load over 25 Satellites 83
4.14 Vela evaluation: Scaling out over a cluster of multicores, Vela running TPC-E 84
4.15 Vela evaluation: Manual online reconfiguration of Vela 86
4.16 Vela evaluation: Target function pseudo-code for Router CPU-idle % 89
4.17 Vela evaluation: Auto-reconfiguring the Router core count based on the Router CPU-idle % target function 90
4.18 Vela evaluation: Target function pseudo-code for Master DPI database reads/second 91
4.19 Vela evaluation: Auto-reconfiguring the Master DPI memory based on the Master DPI database reads/second target function 92
4.20 Vela evaluation: Target function pseudo-code for the system’s re-
response time ........................................................................... 93
4.21 Vela evaluation: Auto-reconfiguring the number of Satellite DPIs
based on the system’s response time target function .................. 94
4.22 Vela evaluation: Per-tenant automatic system reconfiguration in a
multi-tenant deployment .............................................................. 95
4.23 Vela evaluation: Performance isolation in a multi-tenant deployment 96

5.1 ALB overview: Supporting the requirements of ALB ................. 103
5.2 ALB design: ALB interfaces facing the kernel and control system . 106
5.3 ALB design: System architecture and call stack ....................... 107
5.4 ALB design: MySQL/InnoDB buffer pool page structure ........... 109
5.5 ALB design: OpenJDK parallel scavenging heap structure showing
memory partitioning in different generations ............................. 110
5.6 ALB evaluation: Ballooning overhead in MySQL: Conventional vs.
Ballooned query response times ................................................. 118
5.7 ALB evaluation: Ballooning overhead in OpenJDK: Conventional
vs. Ballooned runtime ................................................................. 120
5.8 ALB evaluation: In-flight memory resizing impact on the TPC-H
throughput of two databases across different VMs ....................... 121
5.9 ALB evaluation: In-flight memory resizing impact on the XMark
throughput of two JVMs across different VMs ............................ 122
5.10 ALB evaluation: Improving the end-to-end latency in a two-tier
system ....................................................................................... 124
5.11 ALB evaluation: Reacting to workload changes in a collocated database
setup ....................................................................................... 124
5.12 ALB evaluation: Latency of ALB grow/shrink operations .......... 127
5.13 ALB evaluation: MySQL/InnoDB ballooning operations latency
histograms .............................................................................. 128
5.14 ALB evaluation: OpenJDK ballooning operations throughput his-
tograms ................................................................................... 129

6.1 Virtualization performance evaluation setup: NUMA topology of
test server .................................................................................. 137
6.2 Virtualization performance evaluation setup: Kernel versions, hard
NUMA affinity and kernel support for paravirtualized spin-locks . . 139
6.3 Virtualization performance micro-benchmarks: Bonnie++ configu-
ration options .................................................. 141
6.4 Virtualization performance micro-benchmarks: Sequential block write
performance ....................................................... 142
6.5 Virtualization performance micro-benchmarks: Sequential block read
performance ....................................................... 144
6.6 Virtualization performance micro-benchmarks: Kernel memory ac-
cess benchmarking pseudo-code .......................... 145
6.7 Virtualization performance micro-benchmarks: In kernel memory
copy latencies ..................................................... 146
6.8 Virtualization performance micro-benchmarks: Network performance
varying concurrency, packet size and network bandwidth ........ 148
6.9 Virtualization performance micro-benchmarks: Effects of multiple
VIFs and network bonding ........................................ 150
6.10 Virtualized database performance: main memory storage running
an OLAP workload .............................................. 153
6.11 Virtualized database performance: main memory storage running a
read-intensive e-commerce workload ........................ 155
6.12 Virtualized database performance: main memory storage running an update-heavy e-commerce workload ..................... 156
6.13 Virtualized database performance: local disk storage running a
write intensive OLTP workload ............................... 157
6.14 Virtualized database performance: network attached storage running an OLAP workload ........................................ 159
Bibliography


