ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 75

Systems Group, Department of Computer Science, ETH Zurich

Exploring scalable transactional data processing in a cluster of multicores

by

Ilias Rinis

Supervised by

Professor Gustavo Alonso, Tudor-Ioan Salomie

Autumn Semester 2012

inf | Informatik
Computer Science

Acknowledgments

I would like to express my sincere gratitude to Professor Gustavo Alonso and Tudor-Ioan Salomie, who supervised my thesis. Their guidance and insight was invaluable and determinant for the successful completion and improvement of my work.

I would also like to heartily thank my family and all those close to me for their unbounded support, tolerance and encouragement during the last six months of hard work.

Abstract

Scalability and elasticity are among the most desired properties that characterize modern, large scale, transactional data processing systems. Common tactics for realizing these properties, tend either to sacrifice other important aspects of the system, such as consistency, or to require a complete database engine redesign. Moreover, scalability is challenged in the light of the modern, constantly evolving, multiprocessor architectures, and, in most cases, the existing approaches cannot cope gracefully with these changes. *Multimed* is a database replication system that offers improved scalability and stability, by deploying replicated engines within a single multicore machine. In this thesis, we present a series of extensions of Multimed; the resulting system is a uniform platform for elasticity and scalability to multicores and clusters of multicores. We present an evaluation of the system, that demonstrates how our approach achieves both vertical and horizontal scalability in a cluster of multicores, and the specifications of the elasticity mechanisms that we adopt.

Contents

Chapter 1

Introduction

Current trends in technology have caused increased access to large scale systems, such as cloud architectures, web applications and transactional data processing systems. Such systems often involve large datasets, increased and concurrent traffic, complexity of requests and distribution; for a system to be able to meet these demands and still provide high performance, a required property is *scalability*. Scalability is the property of a system or service to handle an increasing load by expanding its resources. It has been a fundamental problem for database systems, and has been studied thoroughly; however scalability comes with a trade-off against the *consistency* of a system [5].

Common tactics towards scalability frequently employ service replication; in the case of database systems, this translates to replication of the database engines that can serve requests. Replication is a proven technique that enables the parallelization of a service, and thus increased performance and scalability. There have been systems that achieve to offer both scalability and consistency [15, 17] by proposing intuitive system design and functionality.

Before the era of multiprocessors, large scale systems would experience low levels of concurrency within a single machine, and thus little had to be done to achieve scalability. In fact, traditional database systems would optimize queries one at a time, while trying to reduce the disk I/O bottlenecks of the system. At the same time, only few concurrent transactions or threads would exist in the system, and thus it was possible to resolve contention issues with simple synchronization primitives. In the light of the trends in computer architecture though, the problem of scalability becomes even more complicated. Multiprocessor architectures, or multicores, pose difficult challenges to existing infrastructure systems; typically the hardware evolves much faster than large software systems, especially multicore processor architectures, where the number of available cores increases exponentially [11]. In multicores, it is possible to execute larger numbers of transactions at the same time in the system, and the concurrent transactions are bound to compete on resources and synchronization primitives. As the magnitude of concurrency increases, so does the contention and competition on resources. Such effects have been thoroughly studied [9, 10, 17]; the need for a fundamental redesign of existing architectures and synchronization primitives that would prioritize scalability instead of single-threaded performance becomes evident [11].

There has been ongoing research in the study of multiprocessor architecture effects on database performance and scalability; however, the approaches mostly tackle engine-specific design properties and optimizations. *Multimed* [17] is a system based on database replication that achieves improved vertical scalability with

the number of cores, within a multicore machine. In fact, it leverages a replication approach based on load separation for scalability, to solve the problem that multicores pose to databases, inside the boundaries of a multicore machine.

In this thesis, we explore the horizontal scalability of Multimed, by extending its replication model to a cluster of multicores. The result is a system that achieves both vertical scalability with the number of cores, and horizontal with the number of machines in the cluster, by using database replication and load separation.

Additionally, we extend Multimed with *elasticity*. Currently Multimed can only be statically configured; this makes the system inflexible with respect to dynamic load changes, since it is not able to adapt its configuration to changes in the load and resources. Elasticity enables the system to adapt, and also offers a valuable tool for the configuration of the system. There exists no optimal resource configuration for Multimed, since this depends heavily on the load; additionally, the system introduces a large number of configuration knobs, and elasticity enables us to control the most significant ones at runtime, without the need to shutdown and restart the system.

Finally, in order to design a uniform model for the available hardware resources of the system, we use *virtualization*. Virtualization offers an abstract, uniform hardware model to the system, as well as multiplexing of hardware resources; the resulting model allows the system to perceive the multicore machines of the cluster as a large, common resource pool for the system's components. Additionally, virtualization provides us with useful reconfiguration options to our system, towards our goal of elasticity.

Chapter 2

Background

This thesis investigates database replication in virtualized environments, in clusters of multicore servers, as a technique for achieving vertical and horizontal scalability; this work is based on an existing database replication platform, *Multimed* [17]. In this Chapter we present a survey of the realted work, that has inspired Multimed, as well as the present thesis. Additionally, it includes a description of Multimed, covering the important aspects of the system, to establish the background knowledge that is necessary for this work.

2.1 Related work

Database replication has been studied thoroughly and for long, as a means for transactional processing systems to achieve scalability; numerous approaches propose diverse system architectures, replication models, optimizations and implementations towards scalability. Recently there has been a turn in computer system architecture, going from single or few processor computers to multiprocessors; this evolution challenges both the existing database engines, in terms of their performance, and also the existing approaches for scalability. Research has turned towards exploring the reasons why multicore architectures do not blend well with database engines, and towards redefining parts of their architecture and introducing intuitive solutions. This work has been motivated and inspired by these problems and the approaches taken, and this Section presents the most significant influences.

2.1.1 Database replication

Ganymed [15] is a database replication middleware system, that addresses the problem of scalability of database engines in large scale systems such as data grids and dynamic content web applications. Replication is the most common solution for such cases, however it involves trade-offs that are hard to resolve. Ganymed discusses that the most challenging trade-off is the choice between scalability and consistency. For most similar systems, the two properties are mutually exclusive; more interestingly, commercial systems usually opt for scalability, while research groups attempt to combine both by doing compromises in the one or the other. Ganymed achieves to resolve the trade-off by offering both scalability and consistency, without having to resort to the caveats or compromises that other approaches take. Examples include static partition

of the data, explicit specification of the transaction access patterns, concurrency in the middleware level that reduces the concurrency in the engine and SQL parsing for optimization.

Lazy replication is known to scale better than eager replication [6], but imposes problems when it comes to the consistency of the data. Ganymed proposes a convolution of the two models by designing lazy replication model, which offers an eager service to the clients. Ganymed introduces a novel, lightweight scheduling algorithm (*RSI-PC*) that performs Snapshot Isolation with Primary Copy replication. The scheduler works by separating the read-only and update transactions: updates are sent to the master replica (the primary copy), while reads are sent to any of the other replicas, which essentially act as caches. The scheduler provides an eager service to the clients, by hiding the inconsistencies from the clients, and by transparently synchronizing the replicas. The RSI-PC algorithm forwards update transactions directly to the master, and keeps track of the per-row changes of the transaction (known as the *WriteSet* method), as well as a global database version number. Both the WriteSets and the version number are propagated to the replicas. The read-only transactions are forwarded to the replicas, provided that there exists a replica that is already updated to the latest global database version. If such a replica is not available, the scheduler delays the execution of the transaction. If no such delay is desired by the client, the scheduler can default to forwarding the transaction directly to the master. The architecture of Ganymed consists of the scheduler, the master and the slave replicas, a manager console and the client interface. The manager is responsible of handling failures in the scheduler and the replicas, and also of adding and removing replicas from the system at runtime. This enables the scalability of the system with the number of replicas. It has been shown that Ganymed achieves almost linear scalability in the context of the TPC-W benchmark, with the number of machines (hosting replicas) in the system. Ganymed also provides reliability by reacting to failures, and availability.

The work in Ganymed has been mainly influenced by C-JDBC [4], a database cluster middleware, implementing database replication. The limitation of C-JDBC is the fact that it requires duplicating logic from the database engine into the middleware, such as parsing SQL statements and performing table-level locking within the middleware. Postgres-R(SI) [19] takes a slightly different approach to database replication. Postgres-R(SI) provides a replication system offering Snapshot Isolation consistency that is based on eager replication and group communication. It is built as a modification of PostgreSQL. This system provides a decentralized architecture; the clients are able to submit transactions to all the replicas of the system, without the need of a centralized coordinator. When a replica receives an update transaction, it creates a WriteSet which is multicast to all replicas at commit time, using the total order multicast. The total order is useful in the case of conflicts: all replicas execute conflicting operations according to the total ordering to achieve global serializability. This is an approach that achieves to offer Snapshot Isolation guarantees, while being able to scale with the number of replicas, with an increased throughput (to the cost of a higher response time) as compared to single instance databases. However, the decentralized nature of this approach becomes a disadvantage when bursts of update traffic occur in the system; the replicas in such cases are kept busy by multicasting WriteSets and resolving conflicts.

The *DBFarm* System [14] builds up on ideas from Ganymed; it is a cluster based database server with a lightweight middleware layer that achieves stable and scalable performance for hundreds of different database instances. The insight behind this work is that, while there exist several systems that achieve scaling and parallelization of databases, using cluster replication techniques, little is done in consideration of the multitude of applications that leverage such databases. DBFarm addresses the problem of the deployment of a clustered database engine that can still provide significant scalable performance, while at the same time supporting multiple database instances, and thus multiple applications that use them. DBFarm achieves scalability by doing load separation between read and write transactions. The system consists of several

master database servers (one per database instance), that execute the write load of the system and are used to provide reliability and durability. Scalability is achieved by a cluster of *satellite* database servers, that replicate the master database, and that are essentially an unreliable copy that serves read-only transactions. This is the element that provides a high level of parallelization, and thus horizontal scalability with the number of satellites. The system introduces an adapter component for each database server, be it a master or a satellite. The adapter is responsible of forwarding client connections to the correct master (based on the database instance), and consequently forwarding the transaction to either the master server or the satellites of each instance. The adapter has similar functionality to the scheduler of Ganymed; apart from load separation and routing, it is also responsible for WriteSet extraction and propagation to the satellites. DBFarm thus achieves Snapshot Isolation consistency similarly to the method in Ganymed.

DBFarm differentiates its contributions from existing work, since the techniques used in the context of clustered database replication would cause problems for multiple instance support. The overhead of methods such as group communication [19], or the implementation of concurrency control in the middleware layer, instead of the database [4], would impose significant overhead per transaction, maintenance cost and membership and synchronization overhead. DBFarm ultimately provides with a scalable performance, that is only saturated when the master servers of the instances are saturated.

2.1.2 Databases on modern multicore architectures

Database replication, as we saw, is a proven technique that enables database systems to scale out with the number of replicated engines; this landscape has been explored deeply - the works summarized here only present a few outstanding samples. However, recent architectures introduce new challenges to the internal design of existing database engines; traditionally, databases have been designed and optimized under the computer model of only few processors present, which reduces concurrency to a level that allowed databases to follow non-scalable approaches without cost.

Lately there have been several research efforts that investigate how multicore architectures challenge the traditional database engines, how is their scalability affected, and how can existing databases adapt to the evolution of hardware. In [7] the authors investigate the performance of existing database engines on multiprocessor architectures, provide an analysis of the major bottleneck being data cache misses, and discuss guidelines for future database design. In [13], the authors exploit multicore architectures by introducing a database engine that uses helper cores to perform data prefetching, and achieves significant reduction of L2 cache misses and thus improvement in the execution time in the database engine.

The *Shore-MT* Storage Manager [11], is a system that establishes a paradigm of how database engines should leverage the multicore architectures towards scalability. Shore-MT achieves to scale significantly better in a multicore machine as compared to traditional engines; interestingly, the design for scalability in multicores that the engine follows results ultimately to performance improvements even for the single-threaded operation. The implementation of Shore-MT is based on the Shore storage manager, a full-featured modern DBMS, and is the result of a step-by-step optimization of the components of Shore that create bottlenecks for scalability. The optimizations done in Shore-MT, and that enable the system to scale with the number of hardware contexts mainly include the redesign of synchronization primitives; shortening or removing critical sections in Shore's buffer pool manager, log and lock manager, replacing the existing synchronization primitives of the critical sections and eliminating points where the aggregation of a multitude of threads would create bottlenecks, as for example in the case of global mutexes. By explicitly redesigning

the critical components of the system with scalability in mind rather than performance, Shore-MT manages to achieve significant scalability gains and also performance gains even for the case of single-threaded execution.

StagedDB [8] is an attempt to increase the utilization of the multitude of CPU cores in modern multicore computers, by improving the memory access patterns and by optimizing data and instruction locality among all levels of the memory hierarchy. The resulting system demonstrates optimal performance, by minimizing instruction cache misses, analyzing query plans and thus exploiting the locality that potentially lies in both data and computation. An approach in the same direction is seen in [12]. This work presents an optimization layer that processes queries and uses multi-threading to exploit parallelism in queries and join operators. The optimization strategy employed is adaptive; it leverages existing benchmark results to detect performance patterns and optimize accordingly. The approach optimizes single queries at a time before sending them to the database; the approach manages to leverage multicore architectures and increase the performance of the engine. However, there is a processing overhead added in each query for its optimization, in several stages (initialization, pattern detection and application, runtime measurements, etc).

2.2 Multimed

Conventional database engines employ traditional methods for query optimization, that try to optimize single queries at a time, with respect to the disk I/O bottlenecks. The execution of a query assumes that the query runs alone in the system, and this results in interference among queries that run concurrently in the system. This effect was limited in the single CPU architectures; however, in multicores, the higher availability of resources enables more transactions to exist in the system in parallel, and this in turn results in increased interference among queries. This interference (or load interaction) is mainly created by contention on the synchronization primitives of the system; concurrent transactions will access the primitives in parallel, and the more the concurrent transactions in the system, the higher the contention. Additionally, the competition for resources increases as the number of transactions in the system increases. Load interaction can cause significant performance degradation of a database engine in a multicore machine; its effect has been studied deeper in [17] and [9, 10]. The *Multimed System* [17] introduces an approach that solves the problems of load interaction and contention, that most conventional database engines face in the light of the constantly evolving and omnipresent multicore computer architectures, by separating the load with a primary copy data replication model.

Multimed applies established database replication techniques to achieve scalability, while at the same time optimizing the performance of the system in a multicore architecture. It diverges from existing work in this area, in that it does not perform changes in the database engine, nor it relies on query processing and duplicating database logic.

Multimed constitutes a platform for database engine replication within multicore machines. In Multimed, the multicore machine that hosts the system is modeled as a pool of available resources; its replicated database engines are distributed over the available resources in dedicated subsets; in particular, each of the engines is running on a subset of the CPU cores of the system. The primary copy replication model involves a single primary copy of the database (the *master* database) and several copies of the primary database, or *satellites*, of the master. This approach enables Multimed to perform load separation, by assigning the write transactions only to the master, and the read only transactions to the satellites. In this way, Multimed

limits the load interaction and contention, for several reasons: firstly, updates do not interfere with the reads, since they are executed on the master; secondly, even read requests are separated among the satellites, and thus complex and costly queries do not interfere with the rest of the load; finally, the total load is split into smaller subsets that run concurrently on smaller subsets of cores, and thus the competition on resources is reduced. The result is that combining the performance of several engines running on subsets of cores, the system performs better than a single engine using all the resources of the multicore.

No database engine can perform optimally for all possible loads ([18]), and this is also the case with Multimed. The factor that limits the loads that are suitable for Multimed is the multitude of updates within the load, since it is based on primary copy replication, and updates are only executed on the master. This means that, when the updates in the load are the dominant transactions, Multimed's performance will also be bounded by the performance of the master. However, for loads with a majority of read-only transactions (read-mostly loads) Multimed provides substantial performance improvements.

2.2.1 System Architecture

Multimed is a database replication and management system whose operation can be summarized in three main points: the system is responsible for assigning a resource allocation scheme to its database replicas, by mapping its replicas to hardware resources, within its application layer; for performing the actual scheduling and routing of the incoming transactions towards its replicas; and for controlling and performing the overall communication between the system and its clients.

Multimed presents a standard client-server interface to its clients through a JDBC Type 3 driver; thus, the clients are unaware of the internal replicated structure of Multimed. Internally, Multimed implements a primary copy, lazy replication model, that guarantees Snapshot Isolation consistency of the data to the clients. In the context of this work, the replicas are always full copies of the primary database. However, Multimed offers the possibility of configuring partial replication in the system, by assigning sub-sets of the database's tables to each of the replicas, and keeping the necessary state in order to coherently route queries to the replicas that can serve them. Partial replication has been explored further in [17]; the contributions of this thesis are orthogonal to the replication model.

Replication Model

Multimed implements a primary copy (or equivalently, a master-slave) replication model; the system consists of a *primary* (master) copy that is responsible for the execution of write transactions. The read-only transactions (simple queries) are routed to the *satellite* (or slave) copies of the primary database. At the same time, the master is responsible of propagating the changes to the satellites, in the order that the master has executed them.

Multimed guarantees Snapshot Isolation to its transactions. In Snapshot Isolation, each transaction is guaranteed to view a snapshot of the dataset with all the changes that have been committed to it, up to the point when the transaction was initiated. In Multimed, the satellites must apply all of the changes propagated by the master to their replica, in the same order, and they can only execute transactions that have entered the system prior to the latest change they have applied.

Whenever the master node commits a transaction to its primary copy, that modifies the dataset, it encapsulates the changes in the form of a list of rows that have been modified, along with the previous and current version of each row; this structure is called a *WriteSet*. As soon as a WriteSet is extracted by the master[1], it propagates it to all of the satellites, which in turn enqueue the WriteSets in their *WriteSet Repository*. Consequently, a satellite replays the WriteSets that were extracted by the master in the same total order as they were executed on the master, thus creating a consistent view of the database at the beginning of the execution of a transaction.

2.2.2 The Components

Multimed consists of three main components: the *Computational Nodes*, the *Dispatcher*, and the *Communication Component*.

The Computational Node (also referred to as Node) models a set of hardware resources (CPU cores, memory), and a set of software resources that among others (connection pools, queues, etc.) include the database engine and its stored data. Hardware resources include the CPUs and memory dedicated to the replica, and software resources include the actual replicated database and the respective engine, the connection pools, the WriteSet Repository, etc. The Computational Node is responsible of forwarding load it receives from the Dispatcher to its database engine and returning the results to the clients. Both the master and the satellites are modeled as Computational Nodes, however each operates slightly differently. The master is responsible for executing the incoming write transactions and propagating WriteSets to the satellites, while the satellite Nodes execute read-only load and apply the WriteSets they receive from the master. Frequently we refer to the Master Node and the Satellite Nodes to differentiate between the two roles of Computational Nodes.

The Dispatcher is responsible for routing incoming load to a suitable Computational Node. In particular, it binds a transaction of the system to one of its Nodes; this is a transaction-specific binding, and can only be changed if the transaction marks itself as an update. Routing is performed in several distinct steps: if the transaction is an update, it is bound directly to the Master Node. If not, the Dispatcher searches for a suitable Satellite. The suitability of a Satellite depends on two criteria: firstly, a Satellite is considered for routing only if it has already applied all the WriteSets that the Master has propagated and its replica is thus up-to-date. This is achieved by marking each incoming query with a timestamp; if the Satellite has committed all the updates up to that timestamp, it is considered up-to-date. Among them, the Dispatcher selects the Satellite that has currently the least load (the load balancing function can consider various different metrics, such as active connections, CPU load, etc.). While no Satellite can be chosen based on these criteria, the Dispatcher will wait for a small period of time, and then retry the selection (only up to a limit of binding attempts). However, if the Dispatcher fails to select a suitable Satellite (in which case we characterize the Satellites as *impotent*), even after a number of retries, it routes the transaction to the Master, which always has a consistent and updated view of the database.

The Communication Component is an asynchronous server based on *Apache Mina 2.0* that controls the I/O of the system. This component forwards the incoming requests to the Dispatcher, aggregates the

[1] WriteSet extraction in Multimed is implemented using SQL triggers and server side functions, for all *update, delete* and *insert* statements.

results from the Computational Nodes and returns the results to the clients through a constant-sized pool of *Java NIO Threads*.

The design of Multimed allows for several optimizations that can improve the system's performance even further; even though there can be plenty configuration points that significantly affect the system, a few of them can be optimized for the general usage of Multimed.

2.2.3 Contribution and Limitations

Multimed introduces an approach that solves the performance limitation of conventional databases when running on multicores. Within the replicated databases of Multimed, the phenomena of load interaction and contention are efficiently reduced, which results in increased performance and scalability with the cores of the multicore.

Multimed, being a middleware system runs over existing database engines, still naturally introduces overhead in the execution of the queries. As compared to a single standard database engine, Multimed introduces latency in the execution of each transaction; incoming transactions need to be processed in the Dispatcher, wait until a suitable Node is found to execute them, and finally once the execution is completed, the results go once again through the system's Communication Component and wait for an available NIO Thread to return them to the client. The binding procedure can create even higher latency, in the case when the Satellites cannot keep up with the changes in the dataset and the load; the reason is the repetitive procedure of waiting a small delay and retrying to select a Satellite for binding. However, this overhead is easily compensated by the reduction of load interaction and contention among queries, and by the increased availability of resources for the execution of each query.

Multimed's performance improvements in contrast to a stand alone database engine are subject to the characteristics of the load as well. It is impossible to create a system that behaves optimally for all kinds of different loads, and this is also the case with Multimed. As with any primary copy replication model, workloads that are dominated by the presence of update transactions are limited by the capabilities of the master copy. Usually such cases are dealt with assigning powerful engine configurations for the master; Multimed also copes with write intensive loads in a similar way. For workloads that demonstrate an adequate number of queries (and actually, the more complex the better), the performance benefits of Multimed are only limited by the hardware itself. As the system scales up, so does the utilization of the various hardware components such as network I/O, CPU and disk - but once a system reaches the limit of its hardware, it is nowadays inexpensive to upgrade it.

Chapter 3

System Description

Multimed is a system that achieves scaling up with the cores of a multicore machine using single master data replication and isolation of the hardware resources to its replicas. For a wide range of different loads, Multimed can provide with significant performance improvements in contrast with a stand alone database system.

The design of Multimed brings up naturally further directions for the system. In its core, Multimed is a distributed system of database engines; however, until now it has been limited within the boundaries of a single multicore. The existence of multicores within clusters is nowadays a common phenomenon in large transactional processing systems, data centers and the cloud; It follows that Multimed should be able to *scale out* as well, with the number of multicore machines in a cluster. Its design models a multicore as a pool of resources that are distributed among the system's component, and there is nothing to prevent the system of considering a cluster of multicores as its resource pool. During this thesis, we take this further step and deploy Multimed in a cluster of multicores, by distributing its components among the multicores of the cluster.

Scalability is among the most important characteristics of transactional processing systems for modern cloud architectures, as is the elasticity of the system. Both the magnitude of the load that these systems receive, and also the vast availability of resources, necessitate the capability of a system for reconfiguration. Should a system lack elasticity primitives, the over-provisioning of the system is inevitable, and this in turn complicates the system's ability for optimizing its utilization. Our experience with Multimed has shown thus far that it is impossible to tailor the system's configuration perfectly into the requirements of a specific load without cumbersome trial-and-error. Should the load be unpredictable (which is the case in real world system deployments), configuring the system statically and while constantly maintaining high performance is impossible. Reconfiguration at runtime will enable Multimed to react adaptively to the load and match its demands, relieving its configuration procedure of unwanted over-provisioning and resource under-utilization; this is the second main goal of this thesis.

Additionally, we now use virtualization within Multimed; we encapsulate each of its components within Virtual Machines, and deploy the Virtual Machines among the multicores of the cluster. Virtualization is a proven, popular and readily available solution in cloud architectures, that will aid us in providing a uniform abstraction and multiplexing of the hardware resources of the cluster. Moreover, virtualization provides a series of appealing opportunities for resource management, that fit well into our desired profile of Online

Reconfiguration.

3.1 Architectural Overview

As seen in Section 2.2, Multimed is actually a middleware server, consisting of a Dispatcher component and a set of Computational Nodes that abstract the set of hardware and software resources that correspond to a replicated database of the system. The new system architecture embraces the same design as its predecessor, with several additions. The new system does not only distribute the database replicas over sub-sets of resources of a single machine, but rather over sub-sets of the resources of a cluster of multicores; the resource pool of the distributed platform is composed of a cluster of multicores, instead of a single multicore. In practice, now Multimed consists of a set of physical machines that each hosts one or more of the building blocks of the system. Examining Multimed's architecture, the distribution formula is natural: the various Nodes can be independently distributed over the cluster, and so does the Dispatcher. Moreover, we introduce a new type of Computational Node in the system, the *Hot Backup Node*.

The Hot Backup is a special form of a Satellite Node; the Dispatcher does not route any requests to the Backup; the Backup only receives WriteSets from the Master and replays them on its dataset. The Backup enables the system to keep a constantly updated copy of the dataset, so that when a new Node is added to the system we won't have to replay a huge log of changes to bring the dataset to a consistent state; rather, only a copy of the Backup's data set will give the new Node the up-to-date view of the dataset.

An overview of the architecture of the system, deployed in a cluster of multicores within virtual machines, can be seen in Figure 3.1. The system now models the union of all the resources of the cluster's multicores into a large hardware resource pool, and distributes resources from this pool to the system components. Thus, the placement of components to physical machines is constrained only by each resource requirements of each component. In Chapter 4, we will show a series of different setups and the effect of configuration and placement of components.

3.2 Distribution

Multimed is a database replication system; previously, it was a distributed system of database replicas, spread over the resource pool of a multicore machine. The distribution property, and the design of Multimed with non-overlapping resource affiliation to its components, do not prevent, but rather suggest, the distribution of the components to a cluster of multicores. Much like the single multicore, that used to host Multimed, hosted all the system components, now each multicore of the cluster will host a subset of the system's components. As explained, this is a choice based on the configuration and resource requirements of each component. However, the design that we now introduce to Multimed, handles all multicores of the cluster uniformly; provided that any two multicores have enough resources for the configuration of a component, it should make no difference whether the component will be placed in the one multicore or the other.

The distribution over a cluster of machines does not change the way that the system components communicate with each other. Previously in Multimed, the Dispatcher would communicate with the replicas

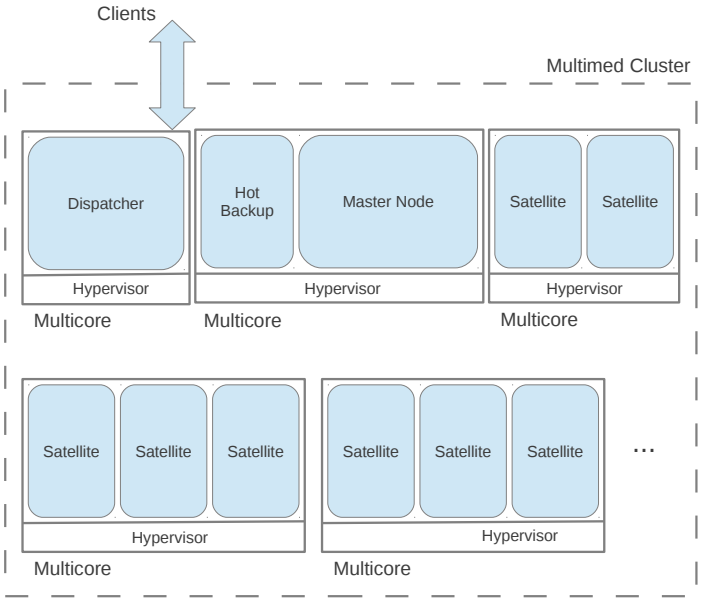


Figure 3.1: Architecture overview of Multimed deployed on a cluster

through PostgreSQL JDBC connections; thus the change in design now is seamless, and the distribution and replication remain transparent to the client.

3.3 Virtualization

The new design of the system requires a way to uniformly and abstractly describe all the available hardware resources in the cluster of multicores. We have chosen *Virtualization* as the platform to build our system model, as it provides a uniform model of the hardware resources and hardware multiplexing; at the same time, it is a readily available technology, proven in the cloud architectures. Last but not least, as we will see further on, virtualization provides with intriguing features for system reconfiguration, that will aid us in enhancing Multimed with Online Reconfiguration mechanisms.

The main component of Xen's architecture stack is the *Xen Hypervisor*. The Hypervisor is a thin software layer, running on top of the host machine's hardware, responsible for the management of CPU, memory and interrupts. On top of the Hypervisor, a specially privileged Virtual Machine (also called a *Domain*) - the *Domain-0 (Dom0 in short)*. The Domain-0 is a control domain that contains the drivers for all the devices in the system, and also a tool stack for the management of the creation, destruction and configuration of the virtual machines.

3.3.1 Xen

For enabling virtualization in Multimed, we use *Xen*. Xen is a Virtual Machine Monitor (VMM) that *"allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacricing either performance or functionality"*(P. Barham et al, [2]); it is an open-source

type-1 (bare metal) Hypervisor, using a micro-kernel design with a small footprint and a limited interface. For this reason, it is considered to be one among the most robust and secure hypervisors [26].

The main component of Xen's architecture stack is the *Xen Hypervisor*. The Hypervisor is a thin software layer, running on top of the host machine's hardware, responsible for the management of CPU, memory and interrupts. On top of the Hypervisor, a specially privileged Virtual Machine - the *Domain0 (Dom0 in short)* - acts as an intermediary between the Hypervisor, the hardware and the *Guest Domains* (the hosted virtual machines). Dom0 runs a Xen-enabled Operating System Kernel and is responsible for managing I/O, as much as for offering a Toolstack and Console control interface to the user.

Xen offers two types of Guest Domains: *Paravirtualized (PV)* and *Full or Hardware Assisted Virtualization (HVM)*. Full Virtualization is a legacy virtualization technique where the Hypervisor, through hardware emulation (Xen uses Qemu), provides platform functionality that resembles a native system. This includes the emulation of CPU, disks, network, memory, interrupts, and so on. In Paravirtualization, introduced by Xen [26], the main idea is to cooperate with the Guest in forming a powerful virtualization environment, instead of having the Guest believe it runs on a physical machine. PV uses no emulation, but rather replaces it by using specially designed interfaces (the *hypercalls*) that allow the Guest to communicate explicitly with the Hypervisor regarding interrupts, memory access, etc. Xen provides with several hybrid modes of Virtualization, that all together constitute a *Virtualization Spectrum* [28] instead of single distinct techniques, that combine the two milestone techniques of HVM and PV. However, further analysis of the Virtualization Spectrum is beyond the purpose of this document.

In Multimed, we use Paravirtualized kernels for the domains that host the various components of the system. The components of the Xen architecture now become essential parts of the architecture of Multimed; most significantly, the Dom0 of a machine hosting Multimed components, is as important as each one of the components, as the reader will find out in Chapter 4. Figure 3.2 shows how Multimed, with the aid of Xen, deploys its elements inside a multicore host machine of the cluster. Dom0 is henceforth considered part of the system; it requires resources that eventually are deprived of Multimed, and, as it turns out, it might occasionally become a sensitive point of overhead.

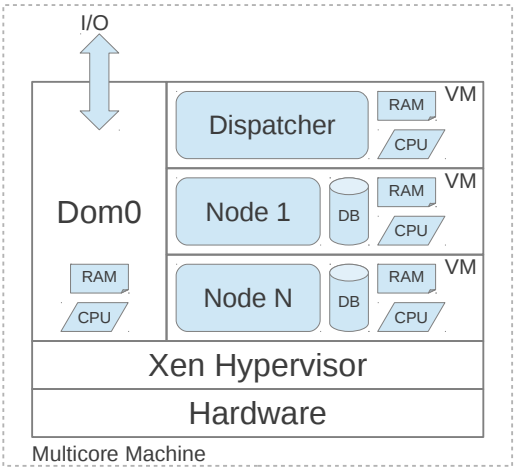


Figure 3.2: A multicore as a host of Multimed's components in VMs

Virtualization brings minor alterations and at the same time new features to Online Reconfiguration. With Xen, Node-level reconfiguration pertains to manipulating the resources of a VM; using *CPU hotplugging*

and *memory ballooning*, we can easily achieve the desired control. At the same time, moving components of the system from one multicore to another is also possible through Xen's *migration* interface. More details are given in Section 3.4.

In Multimed, the Satellites serve only read-only requests and their state (connection pools, queues, data structures) resides within the Multimed Router. In the context of the Satellites, live migration does not make much sense, since a Satellite has no state of its own within its VM other than its network connections; for this reason it can be trivially shut down and brought up using our approach for System-level Reconfiguration. It is far more interesting to consider live migration in the context of the Router and the Master Node, since we demand that their service remains uninterrupted. The Router keeps all the state of the system, and as we will see, it might cause bottlenecks in the system; the Master keeps the dataset and transaction log of the database, which can also turn out to induce overhead to the system. For this reason, these two elements fit well in the profile of live migration.

Among others, this thesis helps clarify the trade-offs and limitations faced when using virtualization for resource management and scaling out that it tries to achieve. The benefits of virtualization and distribution make scaling out possible for Multimed; however this does come at a cost. Virtualization in general brings overhead in such systems, and at times this overhead might scale along with the system's performance. In Chapter 4, we delve deeper into the aspects of overhead and performance.

3.4 Online Reconfiguration

Elasticity is a significant characteristic of large scale systems. A system that provides elastic primitives, can avoid over-provisioning and consequently wasted resources, and can react to the load by adapting its configuration by either adding resources during high load, or by consolidating its components to fewer resources when the load decreases, without service interruption ([1]). We add the elasticity property to Multimed, by implementing online reconfiguration in the system, that will enable it of altering its configuration at runtime in two dimensions: the Node-Level, and the System-Level reconfiguration. The use cases of reconfiguration vary, and so do the options that the reconfiguration mechanism should provide: from adding or removing resources to the system's components, to adding or removing Satellites to the system, or moving them from one multicore to another.

3.4.1 Node-level Online Reconfiguration

With *Node-level Online Reconfiguration*, we enable the system to reconfigure its components, by adding or removing CPU cores or memory to them. Specifically, Node-level reconfiguration will manipulate the resources of the domain (the Virtual Machine) that hosts the component of interest. This is a standard feature with modern Virtual Machine Monitors; in the case of Multimed, this is achieved through Xen's *CPU core hotplugging* and *memory ballooning*. Xen exposes the following interface:

`xm vcpu-set <Domain> <vCPUs>`
: Set the number of CPU cores of a domain.

`xm mem-set <Domain> <Mem>`
: Set the amount of memory of a domain.

Note: *While experimenting with Xen CPU hotplugging on Multimed's components, we encountered bugs in the kernel version used in our experiments. The first is a known issue at the Xen community: the device manager of the Linux kernel,* `udev`*, does not handle correctly the event that is raised when a CPU is hotplugged in a Domain. In fact, the event is caught, but the manager lacks the specific rules that perform the actual hotplugging of the CPU. Additionally, even when the events are properly handled, we found that after hotplugging several CPUs, some of them would stall, and the rest of the CPUs would realize this problem, notifying the user through the system logs. At the same time though, the domain in question would stop responding. However, we have been unable to resolve this issue. We have bypassed the issue by using the Linux program* `taskset`*, a system tool that is used to set or retrieve the CPU affinity of a running process of the system. CPU affinity is a scheduler property that bonds a process to a given set of CPUs on the system. With* `taskset`*, we were able to set the CPU affinity of the desired processes (all the processes that correspond to the database engine), and change it at runtime - thus simulating the CPU hotplugging.*

3.4.2 System-level Online Reconfiguration

There are many cases where simply resizing a Node of the system turns out to be inadequate; an under-provisioned system configuration will behave differently, depending on the properties of the workload, when its Nodes are given more resources than when new Nodes are added to the system. To handle such situations, we introduce *System-level Online Reconfiguration* to Multimed, the operation by which a Satellite can be added to or removed from the system, or moved from one Domain to another, at runtime, without interrupting its service.

Removing a node from the system is a simple operation; it only consists of removing the node from the Dispatcher's data structures, so that no transactions are bound to it and no WriteSets are propagated to it any more, closing all of the node's connections, and finally shutting its database engine down.

Domain migration

For the operation of moving a node from a multicore to another, we leverage the *migration* interface that the Xen Hypervisor exposes, and in fact move the domain of the node from one multicore to another. Xen's migration process can be separated into two modes, *cold* and *live migration*. The first, is a manual procedure that involves several steps, while the latter is an automated and transparent procedure.

Cold migration aims to reduce the total migration time. The domain being migrated stops the execution of all processes and copies its state (e.g. page tables) to the destination machine, where the state and execution are resumed. The interface for this procedure is

```
xm migrate <Domain> <Host>
```

Live migration focuses on minimizing the time that the domain is suspended while migrated. During live migration, the domain's execution does not get interrupted; the memory and network connectivity of the domain are transferred iteratively to the destination, and a final synchronization step completes the procedure before the execution seamlessly continues operation from the new host. The interface for live migration with Xen is

```
xm migrate <Domain> <Host> --live
```

Adding a new Satellite

In contrast with the aforementioned reconfiguration operations, adding a new Satellite to the system is a more involved and interesting procedure. When adding a new Satellite to the system, it is important to be able to bring the new node up-to-date and running as quickly as possible. We achieve this using the existing design and mechanisms in Multimed, with a single addition: we introduce a new, special node in the system, the *Hot Backup* (or simply Backup) Node. The Backup is similar to a Satellite; it is responsible of keeping an up-to-date database replica, just like any other Satellite of the system. However, the Backup does not serve any of the incoming requests of the system. Its sole purpose is to constantly receive updates from the Master, apply them to its dataset and thus maintain an always up-to-date and active copy of the dataset.

The procedure of adding a new node to the system is transparent; the clients are unaware of the background procedure of creating the new Node, and the system's performance will remain unchanged throughout the whole process, until the new Satellite is created and initialized. In more detail, the addition of a new Satellite is broken down to the following steps:

1. The Backup stops receiving updates.
2. The Backup database is stopped.
3. The Backup database is copied to the new Satellite.
4. The Backup's queue of pending WriteSets is copied to the new Satellite.
5. The new Satellite is added to the Dispatcher's data structures, so that it is ready to receive traffic and WriteSets.
6. The new Satellite and the Backup start consuming pending updates. Also, the Master Node starts propagating WriteSets to the Backup and the new Satellite.
7. The new Satellite completes applying all its pending updates and thus starts receiving traffic from the Master.

The addition of a new Node is completed seamlessly, with no service interruption. During step 3, the database of the Backup is shut down, and thus no WriteSets can be replayed. Instead, the WriteSets are being queued in the queue of the Backup; once the copying is completed, this queue is copied to the new Satellite, and after this step the new Satellite and the Backup are initialized and start replaying the pending WriteSets to their dataset (step 6). The Dispatcher will start routing transactions to the new Satellite only once it has completed replaying all the pending WriteSets, thus having a consistent view of the dataset. Note that the copying of the database in step 3, is done with `scp` over the network; the copying of the WriteSet queue in step 4 is an internal copy operation in the process of the Dispatcher.

In Section 4.4, we present a demonstration of the procedure of adding a new Satellite to the system, in order to visualize the behaviour of the system during each step of the procedure.

Chapter 4

Evaluation

Multimed includes several diverse architectural layers in its design: symmetric multiprocessing, networking, virtualization, transactional data processing, and resource management. To combine all these layers into a scalable system is a complicated task; the system introduces a large number of configuration knobs, with an even larger number of dependencies and correlations on each other. Ultimately, it is the understanding and proper configuration of each of these knobs that will enable the system to meet its requirements and goals.

This Chapter provides an evaluation of the system, a procedure that has helped us decidedly to gain a deep understanding of *a*) the performance of the system, in terms of its scalability; *b*) the significant parts of the architecture of the system, with respect to their effect on the system's performance; *c*) the critical system configuration knobs, and the manner in which they can affect the system; and *d*) the optimizations that are (or can be) performed on the system, in order to improve its performance . Eventually, the goal of this analysis is to clarify the contributions of our system, as much as its limitations.

We initially introduce the reader to the parts of the system that can affect its perfromance and operation by creating bottlenecks; we continue to provide a complete description of our evaluation environment prior to presenting the results of the evaluation; finally we conclude by presenting a discussion and analysis of the results, and future directions that the system might follow.

4.1 Bottleneck Analysis and Optimizations

The design process of the system, but more importantly the evaluation of its performance has given useful observations regarding the system's components that might possibly create bottlenecks. The aim of evaluating a system is of course to pinpoint its limitations and attempt to mitigate them; in this section we describe the bottlenecks we have identified through the evaluation process. The bottlenecks will also be demonstrated in context in Section 4.3 where the results and the analysis of Multimed's evaluation are presented.

Each of the components of the system has its own role, and can result in different bottlenecks in the system. Occasionally, resolving a specific bottleneck reveals the existence of another, or affects the system so that another bottleneck is created elsewhere. We explore the way that each system component can create bottlenecks to the system's performance, and present the methods used to mitigate or resolve them.

4.1.1 The Satellite Node

The Satellites are the main processing unit of read transactions. The Dispatcher will offload as many read transactions to the Satellites as possible. Consequently, the CPU cores of the Satellites can reach very high utilization due to the data processing within their database engines. Once the system hits a CPU bottleneck in the Satellites, there are two ways to further increase the throughput of the system, through Online Reconfiguration: one can either add CPU cores to the Satellite, or add a new Satellite to the system. The current design model of Multimed requires symmetric Satellites, in terms of the resource allocation (number of cores and memory). We have experienced that, due to this symmetry, once a Satellite's CPUs are over-utilized, this happens with all the Satellites of the system; thus, we mostly facilitate system level reconfiguration to alleviate the bottleneck of the CPUs of the Satellites.

The Satellites are also constrained on their available memory (especially in the case of virtualization, where this is explicitly defined for each VM - when Multimed runs natively, the overall system memory is available for all components). Memory affects the database engine of a Satellite in several ways. Firstly, caching the data into memory significantly reduces the response time of the transaction processing. For the Satellites, we place the datasets in fast volatile memory, thus achieving a similar effect as if the dataset would be completely cached into main memory. This offers significant a performance improvement for a Satellite, and does not violate the durability requirements of the system; the Master is where the updates are executed and is the Node that has an always consistent view of the dataset - thus it is the point where durability is enforced. For different configurations though, where this is not desired or possible, the Online Reconfiguration of the system allows for memory ballooning to change the memory of the VM that hosts the Satellite, including the database engine's cache.

Moreover, each connection to the database engine requires a certain amount of memory and this can limit the number of connections existent in the engine, which in turn limits the number of active transactions. For some database engines, like MySQL, this is a tunable but static parameter, but for others, like PostgreSQL, the cache of the database freely takes up the necessary space in the system memory. Last but not least, the shared buffers of the database engine define a memory section that is shared between the processes of the engine; in the case of PostgreSQL, this is also the memory region where the engine does its processing. This is typically a static parameter of the engine configuration. Static configurations might create performance issues for Multimed, and the Online Reconfiguration mechanism that we have designed for it does not help in such cases. However, recent work [16] shows methods that Application Level Ballooning can be used to manipulate the memory configuration of such cases.

4.1.2 The Master Node

The load, that the Master Node receives, depends on two factors: the rate of updates in the load, and the existing Satellites in the system. The first factor is obvious; the more updates in the load, the more requests go to the Master. The Satellites of the system affect the Master's load as well; when the Satellites are able to serve the read load, and to keep up with the Master's updates, the Master will receive only update requests. However, when the existing Satellites cannot keep up with the load or the updates propagated by the Master, the result is that no read load is routed to them until they reach a consistent view of the data, and thus it is routed by default to the Master. In any case, this may cause high CPU utilization on the Master. This is due to either the transaction processing in the Master's database engine (both for read and update load), or because of the I/O wait time while the Master writes its updates to persistent storage. To tackle I/O bottlenecks and

further increase the Master's I/O throughput, we use NAS storage for its dataset; NAS storage provides high throughput to the storage, and this can further reduce the I/O bottleneck. When the CPU utilization of the Master is the case, one has two options to resolve the bottleneck: if the increased utilization comes from the inadequacy of Satellites and increased read load in the Master, more Satellites can be added to the system, or more cores can be given to the existing Satellites. In the case where the Master's cores are highly utilized because of its update load, one can give additional cores to the Master.

The memory of the VM that hosts the Master can also cause performance degradation; similarly to the Satellites, the database engine has memory requirements that increase with the load. The same methods can be used to resolve this, as in the case of Satellites, except for the option of placing the database in-memory. The Master must provide durability and thus the database must reside on persistent storage. However, given enough memory, the Master may still be able to fully cache its dataset.

The Transaction Log

Update transactions perform changes to the dataset, and thus invoke writes to the persistent storage of a database. Modern database engines control and mitigate this overhead by using *Transaction Logging* (for example, the *Write Ahead Log* of PostgreSQL [23]). Transaction Logging is a mechanism that enables a database engine to aggregate the row updates, to dedicated files, and periodically flush the content of these files (the Logs) to the actual database files. This approach leverages the benefits of *sequential I/O*, in contrast to *random I/O*. The contents of the log are flushed to the database either after a period of time (the *Checkpoint*), or when the pre-defined log size is reached.

Consequently, the transaction log performs the majority of the I/O operations of the engine. The effect of the log is that the more updates in the load, the higher the log utilization. This can be a bottleneck in the Master Node; the resolution of the issue is not specific to Multimed but rather hardware or engine related. For the transaction log, we do not use the NAS storage that the Master uses for its dataset, but rather place it in a local disk. The high rate of writes to the log may cause a bottleneck; this can be mitigated further by using a faster disk as a host of the Log. In Multimed we have used standard local disks instead of NAS storage, occasionally SSDs instead of standard disks, when necessary, and also RAID arrays (in *striping* level; two or more disk arrays are used to parallelize writes and offer improved write performance). Additionally, the database engine typically offers a range of tunable parameters regarding transaction logging, such as the checkpoint frequency, the size and limit of the size of log files, etc.

Especially for the Satellites of the system, the transaction log and the dataset's storage should be utilized as less as possible; the reason is, as said before, that the Satellites should not offer durability. To minimize the overhead of writing to the log and to the dataset as much as possible, for the Satellites we disable the following options of the database engine:

1. *fsync*. Enabling this parameter will cause the engine to ensure that the data are written to persistent storage before completing each write, by using the `fsync` system call; this process induces delay to a write operation.
2. *Synchronous commit*. When this parameter is enabled, the engine will wait for the log records to be written to disk before completing a commit, and aims to ensure a transaction about the durability of its changes. This operation is unnecessary for the Satellites, and is disabled to avoid delays.

Disabling the two options will increase the performance of the Satellites' engines, while not compromising the durability of the data in the system.

4.1.3 The Communication Component

The Communication Component is the one that handles all network traffic with the clients of the system. Increasing incoming transactions lead to increased number of network connections and packets; the ability to cope well with these depends on the network interface used, and its capacity. This is a factor out of the scope of Multimed; for this reason, we have intentionally reduced the amount of data exchanged by the transactions in order to reduce the payload of the network packets.

Apart from hardware, the number of NIO Threads that Multimed is configured with, affects how fast incoming connections are bound to Nodes, and how fast results are returned to the clients. This is a trade-off though, as increasing the number of NIO threads might cause an increase in the CPU utilization due to the thread scheduling and context switching. We have not thoroughly established the best strategy of choosing the number of NIO threads of the system, as this depends on numerous factors, including the number of cores, the load's characteristics, the networking performance, etc.

4.1.4 The Dispatcher

For every transaction that comes in the system, the Dispatcher performs its routing operation and binds it to a suitable Node (recall Section 2.2.2). Read-only transactions are routed to the Satellites; however if there is no suitable Satellite, the Dispatcher performs number of retries (with a delay in between) until the procedure backs off and routes the transaction to the Master. This is a constant and costly procedure, that increases as the incoming load increases, and may result in high CPU utilization of the Dispatcher. This bottleneck is easily alleviated by increasing the number of cores in the Dispatcher Component.

4.1.5 The Domain-0

Xen's Paravirtualization (PV) mode, used in Multimed, brings improvements in the virtualization performance, and removes the overhead imposed by the emulation of the hardware of the system. Especially for networking, it is claimed that it achieves near-native performance [27]. PV mode creates additional communication channels between the hypervisor and the guest operating systems, called the front-end and back-end drivers. Specifically, for networking, the Domain-0's side of the driver is called `netback` and the VM's, `netfront`, and each network interface of a VM (called a Virtual Interface - VIF), is assigned to and handled by a single `netback` thread. A VIF uses the `netfront` to communicate with the `netback`, which ultimately communicates with the physical network interface card.

We have encountered cases where the performance of PV networking diverges from that of a native system, and where Domain-0 creates bottlenecks in the system. The most network-intensive components of the system are the Dispatcher and Communication Components, since all of the system's traffic goes through them; these components are hosted within the same VM (and in fact, process), and the Domain-0 that corresponds to this VM exhibits very high CPU utilization for larger loads. The VM's VIF is handled by a single `netback` thread in the Domain-0, and large loads will cause this thread to be fully utilized, since the

threads are not balanced among multiple CPUs. Xen does not give optimization options for this; however, we have resolved this problem by creating multiple network interfaces for this VM[1]. These VIFs will then be handled by multiple `netback` threads in Domain-0 as well, and thus, given that enough CPUs exist in the Domain-0, the network load of this VM will be balanced among multiple threads and multiple cores. However, even this fine-tuning is limited by Xen, since it allows the configuration of at most 8 VIFs per virtual machine.

4.1.6 CPU Interrupts

By default, in the Linux kernels we are using (see Section 4.2.1), the CPU interrupts are handled by the first CPU core of the system. Thus, when the traffic is high within a machine (more important for the Domain-0, but also consistent for the VMs that encapsulate the components), the large numbers of network interrupts will cause the first CPU to be over-utilized. To tackle this problem, we balance the interrupts among all cores, by using the program `irqbalance` [20]. This program constantly analyzes the amount of work that interrupts require on the system, and balances fairly and transparently their handling across all the available cores.

4.2 Benchmark Description and Setup

We have designed and conducted a series of experiments on Multimed, that target specific characteristics of our system and that will allow us to make conclusions on its performance. We investigate the scalability of the system, both vertical and horizontal, in both single machine and cluster deployments. Additionally, we aim at exploring the overhead of virtualization in our system, and compare and contrast virtualized deployments with native ones. The evaluation also discusses the various bottlenecks encountered in the system during the experimentation, in order to bring the analysis of Section 4.1 into context. Apart from scalability, we briefly look at the Online Reconfiguration of the system as well, and attempt to demonstrate its operation and usefulness.

The specifics of each experiment, including experimental method, system configuration, etc., are discussed in-context of the description of its results, in Section 4.3.

4.2.1 Hardware and Software Stacks

The main software components of Multimed are the Middleware server, the database engines and the hypervisor. Multimed's Middleware layer is built using *Java 7*, leveraging several of its improved libraries, like the *Java NIO Library*, and the asynchronous server of Multimed is built on the *Apache Mina Server 2.0*. The Computational Nodes are running *PostgreSQL 9.2* for their database engines. The Hypervisor used in Multimed is *Xen 4.1.2*. Both the Domain-0 and the domains that host the components of Multimed, are runing the *Linux Kernel 3.6.6*.

Apart from the main building blocks of Multimed, several other tools are used for optimizations and configuration. The Linux program `irqbalance` is used for balancing system interrupts over all the available

[1] We use interface bonding to simulate the multiple interfaces as a single interface for the needs of IP routing.

cores of a machine; the program `ifenslave` is useful for creating network interface bonds and splitting the network traffic among them; finally the program `taskset` is used to manipulate the CPU affiliation of the various processes of the system (the Multimed main process and the database engines).

The evaluation of the system is done in two distinct classes of benchmarks, and for each class different hardware is required. The vertical scalability evaluation is performed within a single powerful machine, while the horizontal scalability evaluation requires a cluster of multicores.

The Single Machine

The vertical scalability of the system is evaluated by deploying it in a powerful multicore machine. The multicore has 256GB of RAM and an AMD Opteron 6174 processor at frequency 2.2GHz, with 64 (physical) CPU cores.

The Cluster

For exploring the horizontal scalability of the system, Multimed is distributed over a cluster of multicores. Each multicore of this cluster is a machine with 24GB of RAM and an Intel Xeon L5520 processor at 2.26GHz, with 8 physical (hyper-threaded to 16) CPU cores.

The Client machines

Apart from the machines that host Multimed, be it in a single or distributed scenario, the evaluation environment we have used consists also of a set of machines used to act as the clients of the system. We use a varying number of these machines, since every benchmark requires a different number of clients[2]. The machines used to create the system's clients are taken either from the cluster described above, either from a secondary cluster with multicores of 16GB of RAM and an AMD Opteron 2376 processor at 2.3GHz with 8 physical cores.

4.2.2 Benchmarks used

The evaluation utilizes the TPC-W Benchmark [25] to generate the load of the system. Typically, each benchmark creates a suitable number of clients that generate a series of queries and updates, according to the specifications of the TPC-W benchmark. TPC-W is a benchmark that fits well into the profile of Multimed. Inherently, the primary copy replication approaches do not cope well with update intensive loads, and can only handle them by using powerful and resourceful master copies. Multimed can handle such loads in a similar manner; however using an update intensive load fails to demonstrate the scalability of the system, as they result in disk intensive operations and I/O bottlenecks rather than CPU utilization.

For this reason, we have selected the *Browsing* and *Shopping* mixes of the TPC-W benchmark; we have found that the *Ordering* mix has little to offer in terms of the system's scalability, since a load with high rate

[2]We have found that using up to 100 clients per machine that generate load according to the TPC-W Specification keeps the system utilization at reasonable levels

of updates is not well suited for any primary copy replication system. The two mixes used consist of a 10% and 20% of updates in the total mix, respectively, while the Ordering mix contains a 50% of updates.

The TPC-W benchmark specifies both an application and a database level. The focus of Multimed is mainly on the database level, and for this reason we have implemented only this. However, some of the features of the application level, that are required for the correctness of the benchmark, were emulated at the database level.

4.3 Results

The motivation behind the Multimed system relies on the question of how do conventional database engines incorporate the changes in modern multicore architectures, with respect to their vertical scalability with the number of cores. Since the creation of Multimed, there have been advancements in this area, that bring conventional database engines closer to multicore architectures. In order to show that Multimed can still perform better than the existing database engines, we provide with a comparison between the scale up performance of PostgreSQL version 8.3 (which was used and evaluated previously in Multimed) and version 9.2 (which is used in the current version of the system). With this, we expect to show that multicores can still be leveraged in more efficient manner when using Multimed instead of conventional database engines.

Furthermore, the evaluation we perform aims to show how virtualization is going to affect the system. When designing the new version of Multimed, it was expected that virtualization will bring further overhead to the system, since there are extra layers between the software and the hardware. The extra layer of virtualization and the additional internal routing among the various components and their network interfaces, are bound to incur further overhead in the delay that Multimed already introduces at the execution of a transaction. Before establishing the scale out behaviour and benefits of our system, we expect to show that virtualization will pose an overhead; however, we also expect that this overhead will be easily compensated for by Multimed's intrinsic properties.

Subsequently, we need to comprehend and analyze the horizontal scalability of the system, being one of its main goals. We describe and analyze the performance of the system as it scales out to a cluster of multicores, with increasing incoming load. Finally, we demonstrate an example of Online Reconfiguration of the system, in order to clarify the method used and understand its costs.

Throughout the evaluation of the system, we run sets of clients generating the aforementioned benchmarks, and measure the system's performance statistics. In order to comprehend our findings, we constantly monitor the performance and utilization of the machines used, and provide with supporting information based on system statistics wherever deemed necessary.

4.3.1 Discussion on PostgreSQL

The reasons why most conventional database engines are facing a challenge with the modern multicore architectures, is inherent in their design. Most of them employ traditional optimization strategies that optimize single queries at a time, with respect to disk access and disk I/O bottlenecks. When multiple queries are executed in the engine, the approach to deal with concurrency is the usage of threads and the enforcement of synchronization among them when deemed necessary. This results in practically only few of the queries

to run in parallel, and in modern multicores this renders the existence of multiple cores almost useless. Typically, the contention among concurrent queries and the interaction among queries significantly affect the performance of the system, and its scalability with the number of cores.

The latest version of PostgreSQL, version 9.2, was released in September 2012, and among the most important improvements, is its performance and scalability. PostgreSQL 9.2 achieves linear scalability up to 64 cores, and efficient utilization of the hardware resources on larger machines [22]. However, it remains to be seen how comparable are these improvements with Multimed, and in which context. To this end, we perform a scalability evaluation on PostgreSQL 9.2 in comparison with the older version 8.3.

During this experiment, we increase the number of cores available to the database engine, and measure its throughput in transactions per second. The experiment is run for both versions of PostgreSQL under discussion, with identical configurations, on a 23GB dataset, created according to the specifications of TPC-W; we repeat the experiment for low and high load. It should be noted that, each database egnine is deployed in the single multicore of our environment, thus giving us the possibility of measuring the system's performance up to 64 cores.

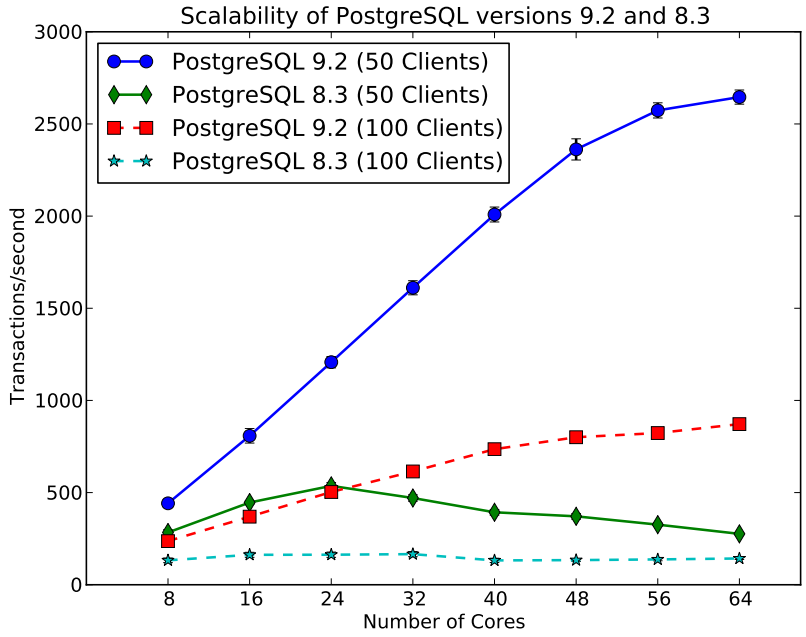


Figure 4.1: Scalability comparison between the two versions of PostgreSQL, 9.2 and 8.3

Figure 4.1 shows the outcome of the experiment in terms of the engine's throughput against the number of cores affiliated with it; the solid and dashed lines correspond to the two different loads, of 50 and 100 clients respectively. This result shows the considerable difference in performance between the two versions: for the load created by 50 clients, version 9.2 scales linearly up to 50 cores; from this point onward, the system is under-loaded and thus its throughput flattens out. On the contrary, version 8.3 initially exhibits a very limited scaling performance, that saturates very soon, only at 24 cores. When increasing the load to 100 clients, we observe that neither of the system tolerates the load. Version 9.2 still scales up with the number of cores, but sublinearly, with a small rate. Version 8.3 in this case does not scale at all, and its performance is severely limited.

Observation shows that the linearity in the vertical scalability of PostgreSQL 9.2 is guaranteed as long as there is a *1-to-1* ratio of connections to cores; simply, each core must handle at most one connection in order for the performance to scale. Albeit its improvements, PostgreSQL still lacks the ability of scaling with the number of cores while the load increases; we now proceed further to demonstrate that Multimed indeed does so.

4.3.2 Vertical Scalability

The set of experiments exploring the scalability of Multimed up with the number of cores in the system aims to *a*) compare Multimed's scalability with that of a conventional database engine in a multicore machine, and *b*) show the overhead that virtualization will induce in the scalability of Multimed, which guides the design the horizontal scalability as well . To this end, we carry out an experiment with a procedure similar to the one of Section 4.3.1. The experiment is repeated for a set of different system variations, all of which are hosted in the single machine of our evaluation environment. The four different *Systems Under Test (SUTs)*, in more detail, are:

Native PostgreSQL. The SUT consists of a single PostgreSQL database engine, that stores a dataset of 23GB, according to the TPC-W specification.

Virtualized PostgreSQL. The same database engine of the native PostgreSQL SUT is deployed within a Virtual Machine in the same physical machine.

Native Multimed. A deployment of Multimed in a native environment within the single multicore machine. The system replicates the dataset of 23GB among its Nodes.

Virtualized Multimed. The SUT in this case consists of an instance of Multimed with its components encapsulated in Virtual Machines, deployed within the multicore.

The initial configurations, in terms of resource allocation, of each of the four SUTs, are displayed in Table 4.1.

Component	CPUs	RAM
Database Engine	8	250GB
Domain-0	2	2GB

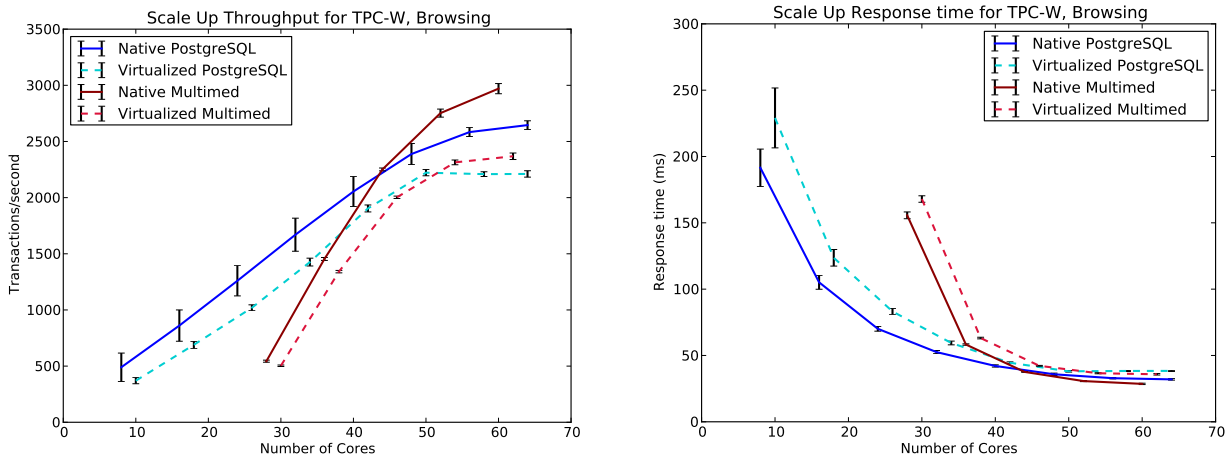
(a) PostgreSQL SUT Variations

Component	CPUs	RAM
Dispatcher	6	25GB
Master Node	14	25GB
Hot Backup	1	25GB
Satellite Nodes	8	35GB
Domain-0	2	2GB

(b) Multimed SUT Variations

Table 4.1: The various SUT configurations for the vertical scalability evaluation

Note that in Table 4.1, the RAM allocated for each component applies only to the virtualized system variations, since in the native systems, the components take up any amount of memory needed from the total available system memory. Similarly, the Domain-0 configuration applies only for the virtualized system variations. We base the choice of configuration of the system on experience; for example, we have seen that for this dataset and benchmark, 14 cores on the Master are enough for the computation requirements. At the same time, the core of the Hot Backup overlaps with the Master's cores, the reason being the fact that



(a) Throughput

(b) Response Time

Figure 4.2: Vertical scalability of the various SUTs (Browsing mix, 50 clients)

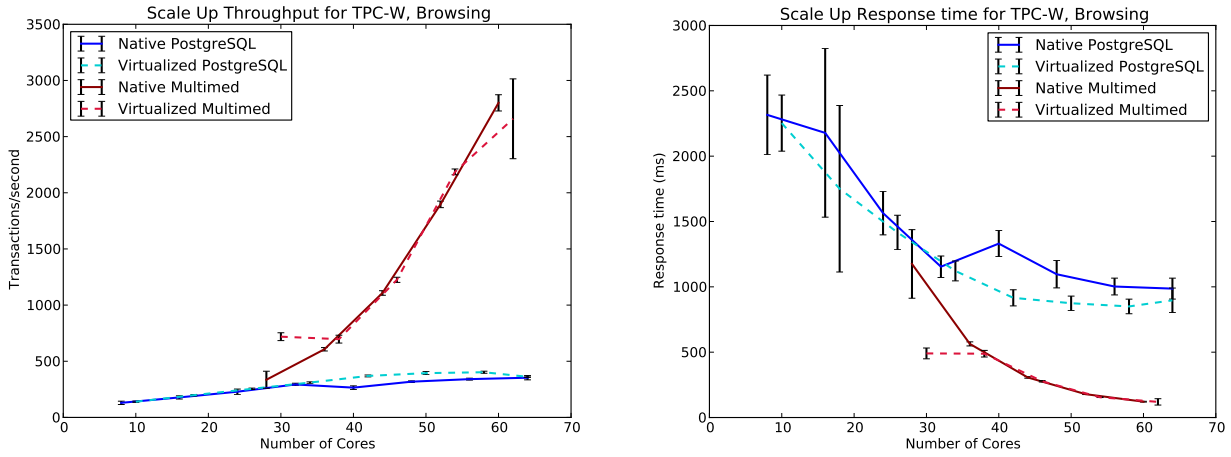
the Backup's CPU is always almost completely idle, except when it is copying its dataset to a new Node. In general we try to create configurations that allocate the minimal amount of resources that achieve the highest performance possible.

Additionally, note that the Dispatcher Component belongs to the same process as the Communication Component; henceforth, we will only refer to the Dispatcher Component, but will imply both.

TPC-W, Browsing Mix

We begin by presenting the results of the experiment regarding the Browsing mix of TPC-W.

Figure 4.2 shows the outcome of the experiment, for a small load of 50 clients. For this experiment we vary the number of cores affiliated with the SUT and measure throughput and response time, for each of the four system variations described. Among the four lines, there exist slight gaps that belong to the different configurations of the system, as seen in Table 4.1. The virtualized variations for example require two additional cores for the Domain-0 than their native equivalents. The linear scalability of PostgreSQL can be seen again in this experiment, for both variations; however, Multimed scales linearly as well, and actually outperforms the corresponding PostgreSQL SUT at around 40 cores for the native, and 50 cores for the virtualized variations. The two native systems demonstrate a degradation of the scalability at about 40 to 50 cores. The reason behind this behaviour is simply the fact that the load of the systems is not enough to further increase the throughput (equivalently, to further reduce the response time) of the systems. On the contrary, the virtualized system variations are unable to reach the same performance; their throughput flattens out at around 50 cores. The reason for this bottleneck has been identified to be the network overhead of virtualization. A series of micro-benchmarks has revealed that network traffic that includes a VM to either of the endpoints receives a constant latency overhead of $0.1ms$ on average, for each network roundtrip. If we also consider the fact that each transaction of TPC-WB has on average 15 requests, we end up with an aggregated constant latency overhead, which effectively increases the minimum response time that the system can reach. Especially in the case of Multimed, where the Nodes but also the Dispatcher reside in VMs, this is a considerable



(a) Throughput

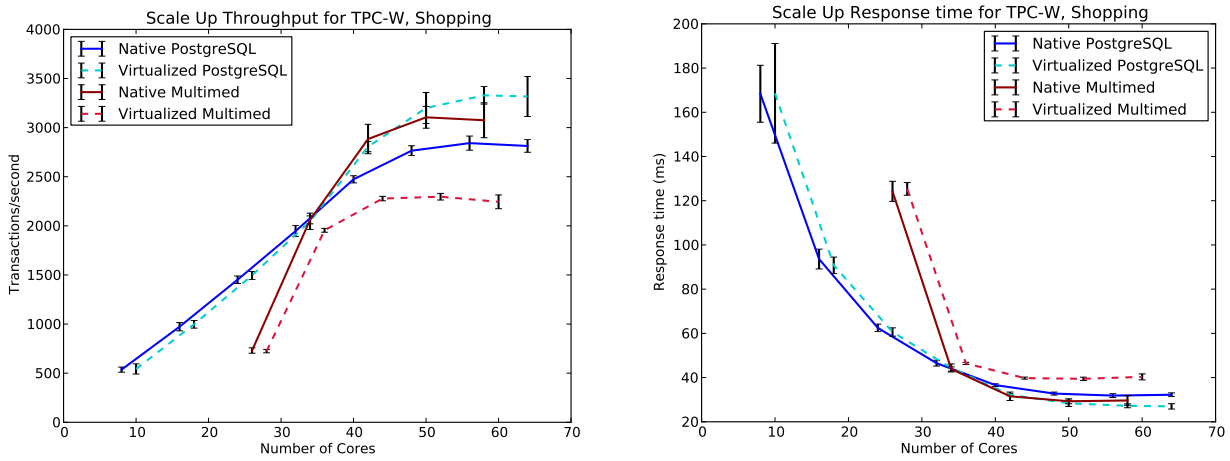
(b) Response Time

Figure 4.3: Vertical scalability of the various SUTs (Browsing mix, 200 clients)

overhead. In general, one can observe that the overall overhead of Multimed is apparent for small loads and for system configurations with few cores (up to 40 cores, a standalone PostgreSQL database engine outperforms Multimed). This is expected, since Multimed can only perform better than a standalone engine when the load interaction and contention among the queries increases significantly, and this only happens for larger loads.

Figure 4.3 shows the results of the same experiment, for the same system variations and configurations, but for a larger load of 200 clients. The most important observation by this result, as expected by the analysis of Section 4.3.1, is that for a load four times larger, PostgreSQL does not scale at all. The fact that PostgreSQL is negatively affected by the increased load can be observed in the response times of the system: while the throughput remains low in a stable fashion, the response time exhibits large fluctuations, as seen in the standard deviation bars. Multimed, on the other hand, scales up linearly with the number of cores. It is only in the full system utilization of the virtualized variation (near 64 cores) that the throughput shows an unstable behaviour (as seen in the standard deviation); the reason is, as explained in Section 4.1.5, the Domain-0's CPU utilization that reaches its limits with a large load of 200 clients.

Regarding virtualization, the latency because of networking is not encountered in this case; this is expected though, since the response times are an order of magnitude larger than the case of 50 clients. Nevertheless, in the case of increased load, there are still several issues that have not been completely understood yet. For example, the first point of the throughput and response time (around 30 cores) between the two Multimed system variations, demonstrates a curious behaviour: there is higher throughput in the virtualized system than in the native. Similar questions raises the performance of the native PostgreSQL compared to this of its virtualized variation. Both the throughput and response time of the system are higher, and lower, respectively, after 30 cores, for the virtualized variation; the performance difference is slightly off the margins of noise depicted in the standard deviation.



(a) Throughput

(b) Response Time

Figure 4.4: Vertical scalability of the various SUTs (Shopping mix, 50 clients)

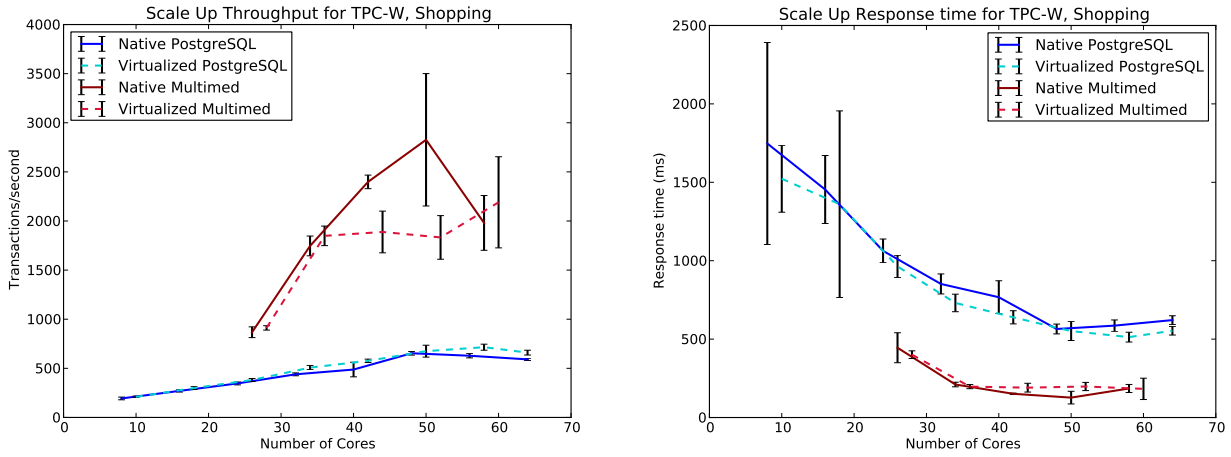
TPC-W, Shopping Mix

The Shopping mix of the TPC-W benchmark has an increased number of updates in its load (20%). Our experience with the Shopping mix shows that the increased update rate results in a more unpredictable and unstable behaviour of the system. We have not yet fully clarified some aspects of the behaviour of this load; however, we present the results, observations and conclusions we have come to, so far.

Figure 4.4 presents the result of an experiment with a small load on the four SUTs described in the beginning of this Section. The experiments are similar to the ones run for the Browsing mix. In the case of the Shopping mix, and for a small load generated by 50 clients, we can still observe the linearity in the scalability of PostgreSQL, but also Multimed. At about 3000 transactions per second, all of the systems start to saturate their performance; the Satellite utilization reveals that the system is underloaded, while at the same time the utilization of the Master reaches very high levels. However, an interesting and curious observation is that the performance of the virtualized variation of PostgreSQL outperforms its native equivalent. The reasons that this happens in the Shopping mix are still unclear to us. Multimed exhibits a similar behaviour as in the case of the Browsing mix; it outperforms the native PostgreSQL system, and there is evident overhead in the case of virtualization. In fact, this time the gap between the performance of the two Multimed system variations is even larger. The reason for the difference in performance is the network latency introduced by virtualization. The Shopping mix has double the percentage of updates, and thus results in double the number of writes to the Master's log, and also to persistent storage over NAS. The higher amount of updates results in higher utilization in the Master node, especially regarding disk operations; this is the reason of the large fluctuations in throughput for higher throughput values. The difference in performance can be also observed consistently in the response time as well; the virtualized deployment of Multimed, after the point of two Satellites in the system, has twice as high response time as the native system.

We push the system even further for the Shopping mix, generating load with 200 clients; the results are shown in Figure 4.5.

PostgreSQL's performance is similar to the corresponding case of Browsing mix; the scalability of the



(a) Throughput

(b) Response Time

Figure 4.5: Vertical scalability of the various SUTs (Shopping mix, 200 clients)

system is significantly degraded. Multimed, on the other hand, again demonstrates increased performance and scalability, accompanied though by an unstable and unpredictable behaviour. The native system scales linearly up to almost 3000 transactions per second, while four Satellites and the Master exist in the system. At this point the throughput of the system exhibits excessively large fluctuations, and then suddenly drops to a much lower value (around 2000 transactions per second). Again, observing the Master utilization, we observe almost full CPU utilization and high log utilization values, that reflect the fluctuations and drop in throughput. The virtualized variation of Multimed scales linearly up to the second satellite; from this point the throughput gets very unstable as seen by the error bars. The unstable performance behaviour for Multimed starts for both systems approximately at the same levels of throughput; the high load in the system in combination with the load that eventually is executed in the Master result in this behaviour. Taking into consideration the error margins of the throughput, we observe that both Multimed SUTs perform most of the time equivalently, both in terms of throughput and in fluctuation.

The Shopping mix seems to push the limits of the system further, especially for large throughputs. Even though the difference in update rates between the Shopping and the Browsing mix is small (20% and 10% respectively), it seems that it is enough to cause instability in the performance of the system. We have not experimented enough with the Shopping mix in order to realize the optimal system configuration and fine-tuning in order to achieve the best possible results in terms of scalability and performance. Additionally, we need to understand in further detail the overhead of virtualization in such cases, since it seems that in the case of the Shopping mix, more factors that affect the system's scalability are revealed.

4.3.3 Horizontal Scalability

The set of experiments run on the single multicore of our evaluation environment have given useful insight regarding virtualization and its overhead. Care must be taken when configuring the Domain-0 of the VMMs, especially for the one that will host the Dispatcher and Communication component. We have seen that for larger loads, the Domain-0 can affect the performance of the system with its high CPU utilization, and

for this reason, before conducting experiments on the horizontal scalability of the system, we perform the optimizations described in Section 4.1.5 and 4.1.6: we balance the CPU interrupts of the Domain-0 among all its available CPUs, and we create multiple VIFs for the VM that hosts the Dispatcher and Communication Component, in order to balance the traffic over multiple `netback` threads, and thus over multiple cores. To achieve this we also have to increase the number of cores dedicated to the Domain-0 of the VM in question, since the more the CPUs, the more the load balancing of the `netback` threads.

These optimizations will enable us to evaluate the horizontal scalability of the system, when the system receives a range of low to high loads. We use the TPC-W benchmark for the load generation, and in particular the Browsing mix. The reason why we chose only the Browsing mix should become clear at the end of this analysis. Regarding the SUT of this set of experiments, we deploy Multimed with virtualization, over the multicores of the cluster of our setup (Section 4.2.1). The multicores now are more limited in resources as compared to the powerful multicore of the previous experiments; in order to be able to deploy large numbers of satellites (up to 25) we choose a smaller dataset, of 4.5GB, in order to be able to still use the model of the system where the Satellites store their datasets in main memory.

We investigate how our system scales out with the number of Satellites (and thus, multicores), by increasing the number of Satellites in the system, while measuring the throughput and response time of the system; we repeat this procedure for a variety of loads. Prior to analyzing the results of each experiment, we also discuss the configuration used.

Basic Configuration

We have seen that the Domain-0 of the VMM hosting the Dispatcher and Communication Component can achieve high CPU utilization for large loads. During this experiment we will use a smaller dataset, and will put larger loads in the system; for this reason, combined with the respective optimizations, we provision this Domain-0 with 6 cores. Moreover, we expect to scale the system out and reach high performance in terms of throughput (and low response time), and so we provision the Dispatcher with 10 cores, in order to cope with all the traffic. For the same reason, the Master node is also expected to receive high load; thus we assign 10 cores to it. The SUT now consists of multiple physical machines, and each hosts a different set of system components, each having different resource requirements. There are three types of configurations, based on the components that each machine hosts: *a*) Satellites-only Host; *b*) Dispatcher Host; and *c*) Master Host . Each different distribution of resources among the system's components is described in Table 4.2.

As we will see, there are still improvements that this configuration can take. Before reasoning about this though, we begin the investigation on the system's horizontal scalability with the number of Satellites, with the first experiment; the load of the system is generated by 200 clients, and the outcome is shown in Figure 4.6.

Starting from 2 Satellites in the system, Multimed scales out linearly with the addition of Satellites. However, from the point where 8 satellites are in the system onward, we observe that the throughput of the system starts to saturate. The same behaviour is observed in the response time as well, where it flattens out at the same point of 8 Satellites. At this point, the throughput is about 6000 transactions per second, and the Dispatcher is handling an amount of network traffic that increases its CPU utilization close to 100%. This is because of the large load on the system, combined with the number of Satellites; the more Satellites in the system, the more work for the Dispatcher, and this only deteriorates with the multitude of transactions

Component	CPUs	RAM
3 Satellites	4	7GB
Domain-0	2	2GB

(a) Satellite Host

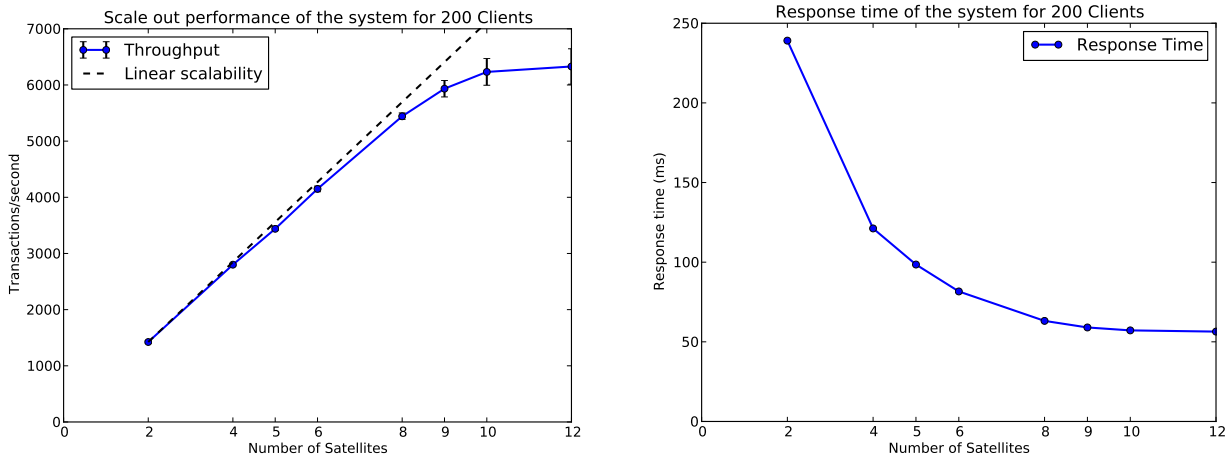
Component	CPUs	RAM
Dispatcher	10	22GB
Domain-0	6	2GB

(b) Dispatcher Host

Component	CPUs	RAM
Master Node	10	14GB
Hot Backup	4	7GB
Domain-0	2	2GB

(c) Master Host

Table 4.2: Distributed Multimed, Basic System Configuration



(a) Throughput

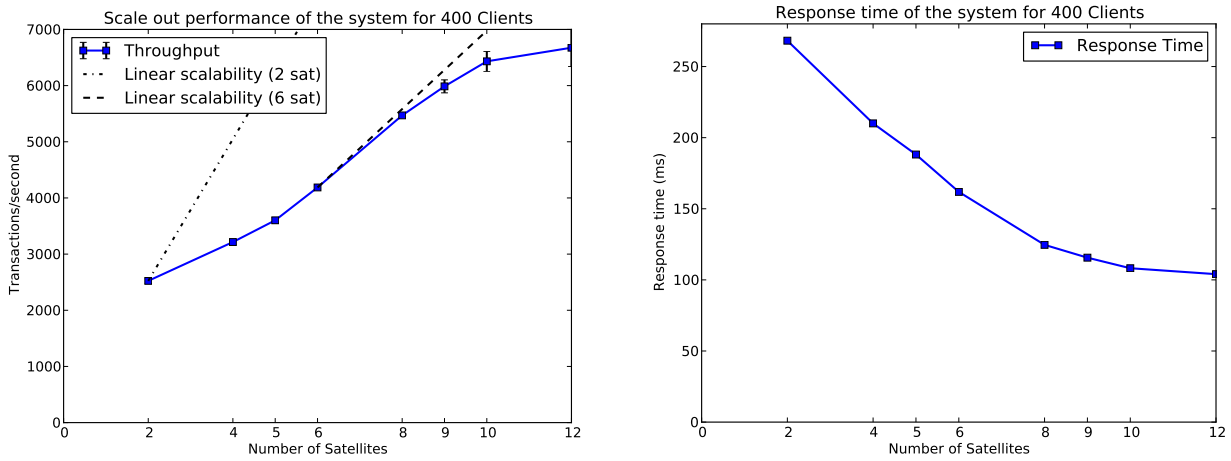
(b) Response Time

Figure 4.6: Horizontal scalability of Multimed (200 clients)

in the system. In order to further investigate this behaviour, we repeat the same experiment, but increase the load of the system to 400 clients. The outcome is presented in Figure 4.7.

The increased load that now exists in the system, causes a sublinear scaling performance when there are less than 6 Satellites in the system. The reason that the system cannot reach linearity in its scalability is the Satellites; for this load, even 5 Satellites are over-utilized. However, when 6 or more Satellites are on the system, their combined computational power can handle the incoming load well; we have observed that from this point, the CPU utilization drops from the absolute 100% continuously as more are added to the system. Starting from 6, and up to 8 Satellites, the system reaches linear scalability; however, at 8 satellites, the throughput is close to 6000, at which point we observe a degradation of the scalability of the system. The reason is the same effect as encountered in the case of 200 clients; the Dispatcher is heavily loaded, and its CPUs are over-utilized.

This does not represent well the limits that our system can reach while scaling out; it is trivial to resolve the bottleneck of the Dispatcher, by giving it more computational power. We alter the configuration of the



(a) Throughput

(b) Response Time

Figure 4.7: Horizontal scalability of Multimed (400 clients)

system, giving more cores to the Dispatcher, in an attempt to explore further the horizontal scalability of the system.

Improving the Configuration

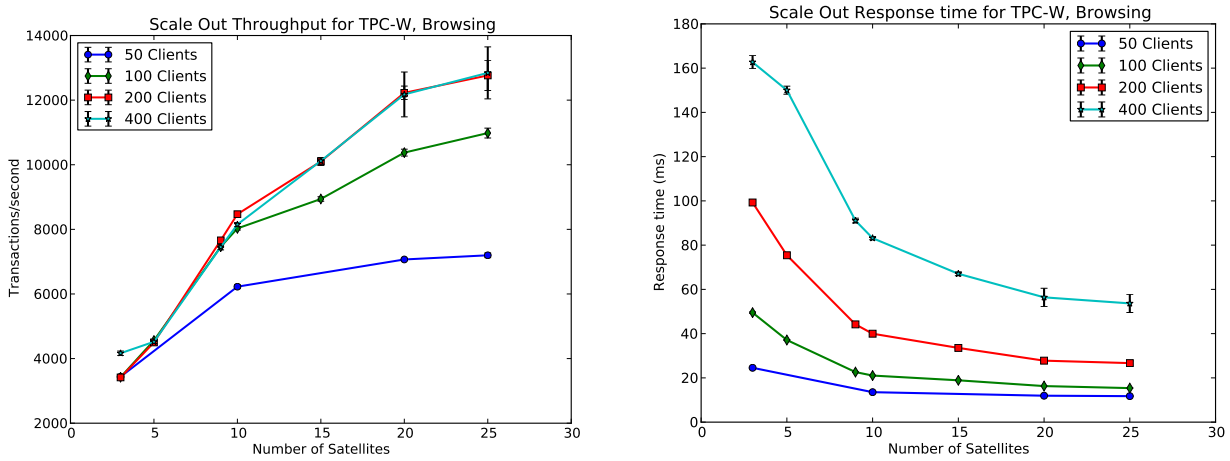
To improve the system performance further, and alleviate the bottleneck from the Dispatcher's CPUs, we change the configuration of the system, seen in Table 4.2. The only alteration in this configuration is that we now place the Dispatcher in the single multicore machine of our environment, that has an abundancy of CPU cores (64) so that the Dispatcher may utilize as many as needed. In fact, during this experiment, we have experienced that the Dispatcher never used more than 16 cores out of the 64; additionally, these 16 cores were not fully utilized even when the system was under the largest load of the experiment.

To avoid further bottlenecks regarding the Dispatcher, we place the component natively in the machine (i.e. no longer within a VM), since we already know how to resolve cases where the bottleneck is located in the CPUs of the Dispatcher or the Domain-0. Additionally, we now expect to reach a higher number of transactions that can be executed in the system per second, and this also implies a higher number of update transactions. For this reason, we use an SSD disk for the Master's transaction log. The configuration is otherwise left unaltered; the change is summarized in Table 4.3.

Component	CPUs	RAM
Dispatcher	64	256GB

Table 4.3: Distributed Multimed, Altered Configuration

Figure 4.8 demonstrates how the performance of the system scales, as we increase the number of Satellites in the system. Starting from 3, we bring up to 25 Satellites, and measure the throughput and response time of the system, for four different loads, in terms of magnitude. Putting a load in the system, generated by 50 clients, causes the system to scale out, until 10 Satellites are in the system. At this point the system is



(a) Throughput

(b) Response Time

Figure 4.8: Horizontal scalability of Multimed (multiple loads)

under-loaded, the Satellites and the Master are not fully utilized, and thus the performance of the system stops increasing.

When extra load is pushed in the system (100 clients), we observe an increase in the performance closer to the linear, since the system with 5 Satellites has a throughput of 4000 transactions per second, approximately, and with 10 Satellites the throughput doubles. At this point, the performance goes up to 8000 transactions per second. Beyond this point though, one observes a reduction in the rate that the throughput increases, and the scalability diverges from linearity. We have identified that in this case, the Transaction Log of the Master has an increased utilization, as compared to smaller throughputs, of about 30%. However, this is not high and should not cause bottlenecks in the system.

We investigate this issue by putting even larger load to the system, generated by 200 and 400 clients. The throughput of the system progresses almost identically for the two loads, except for the starting point of 3 Satellites. The reason is that, at this point, the Satellites are heavily loaded, and for this reason they cannot keep up with the updates they receive from the Master. The result is that the Dispatcher does not route transactions to the Satellites, while they are not up-to-date, and by default all these transactions end up in the Master. In turn, the Master executes the transactions and thus aids the system and increases its performance, as compared to the case of 200 clients, where the load is handled by the 3 Satellites. After the starting point, the two lines converge, and again at the same throughput of about 8000 transactions per second, the same change in the slope is observed. However, this time the throughput is pushed higher, up to 12000 transactions per second; the transaction log utilization in the Master has now gone up to 70%. We observe two issues: firstly, the throughput between 200 and 400 clients reaches exactly the same limits, and the transaction log utilization is similar in both cases, up to 70%; at the same time, the response time of the system for 400 clients is double the one for 200 clients. Both observations prove that the system cannot be pushed further, and is experiencing a bottleneck - for this reason the response time is increased while the same number of transactions are executed per second.

The bottleneck in this case is the transaction log of the Master, even if it has not reached its full utilization. PostgreSQL uses a process that writes data to the log, the *Write Ahead Log Writer (WAL Writer)* process,

and to our knowledge, the number of WAL writers is not a tunable parameter of the system. In the case of this experiment, there is only a single WAL writerin the database engine of the Master; this process is performing approximately more than 1200 writes per second to the log (considering that the Browsing mix consists of 10% updates), and this is the limit that this process can handle. Since this is not a parameter that can be modified in PostgreSQL 9.2, the solution for this case is to use a disk that can achieve even faster writes for the transaction; for example, an RAID-0 array (striping) of SSDs could be used for the transaction log, instead of a single SSD that we are using now.

Our current setup did not allow us to perform this optimization at the moment; we claim though that this will enable the system of scaling even beyond 12000 transactions per second. For the same reason we have not yet performed this set of experiments with the Shopping mix of the TPC-W benchmark. The Shopping mix load consists of 20% of updates; this means that theoretically, one would expect the same transaction log bottleneck to appear in the system, only much faster, since at about 6000 transactions per second, the WAL Writer would perform about 1200 writes per second to the disk. Additionally, the results presented in Section 4.3.2, show that further instrumentation and analysis of virtualization and configuration is necessary for this mix.

4.4 Online Reconfiguration

Section 4.3 has revealed numerous details regarding the design, implementation and configuration of the system, and has demonstrated how the system can achieve high performance and vertical and horizontal scalability. The various components of the system each have different significance in the system's performance and different sensitivity to configuration parameters and overhead from virtualization. One of the most important outcomes of this analysis is the importance of configuration; we have encountered numerous cases where we expected the configuration of the system to be optimal, but in practice realized that it should be altered. Support for Online Reconfiguration for a system like Multimed should now be clearly justified; this Section presents a demonstration of how is Online Reconfiguration achieved in practice, and what is its overhead.

We have already presented, in Section 3.4, the methods used for each operation of the Online Reconfiguration, both in the Node Level and in the System Level. The only non-trivial procedure is the addition of a new Node to the system, and in order to comprehend this procedure in depth, we perform this operation while monitoring the system over time and demonstrate the result. We use a virtualized Multimed deployment in the single multicore machine of our environment. The Nodes of the system replicate a dataset of size 23GB, and the load is generated according to the Browsing mix of the TPC-W Benchmark. The system is started with the Master Node only, and after a period of time two Satellites are added to the system.

Figure 4.9 presents the system state during the addition of a new Node. In this experiment we monitor the system's throughput and CPU utilization over time. There are several vertical lines in the figure; the solid lines denote the point in time when the operation of adding a new Satellite is initiated, and the dashed lines denote the time when the new Satellite is initialized and starts its operation. The main factor in terms of time consumption of the operation is the copying of the database of the Backup to the new Node; this time is essentially the period between a solid line and the next dashed one. During this time, one can see that the Hot Backup is fully utilized, because of the copy operation. The Satellite exhibits also a utilization of 10% for the same reason. Once the copying is completed, the Backup returns to its stable state, which is almost

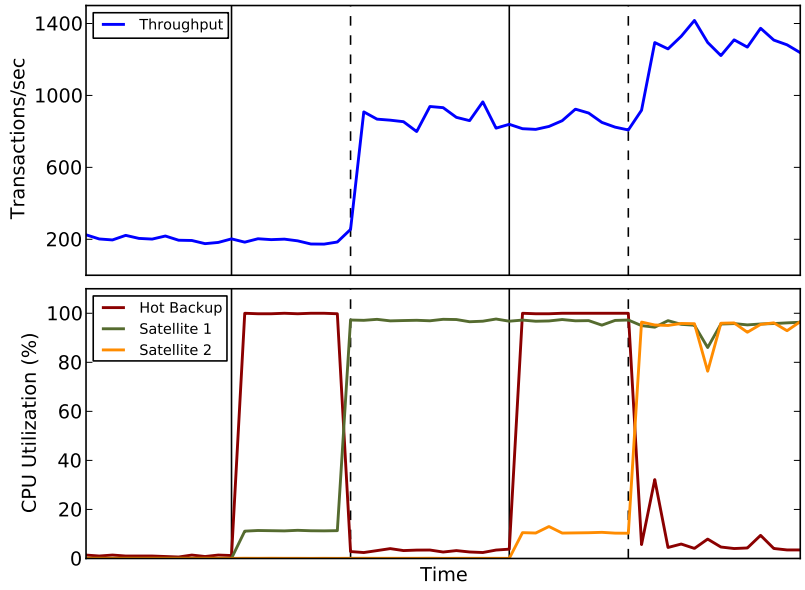


Figure 4.9: Demonstration of System Level Online Reconfiguration

completely CPU idle, the Satellite immediately starts receiving load and processing, and thus its utilization goes close to 100%, andfinally, the throughput of the system increases, due to the new Satellite. The same procedured is repeated after a short period, and a second Satellite is added. During the stable periods of the system we can observe how the Hot Backup's utilization demonstrates a slight increase. During the time when only the Master is present in the system, the Backup is almost idle; the reason is that the Master is receiving both update and read only transactions, it is fully utilized, and the number of updates it executes is small. However, when one Satellite is added to the system, the Master performs mostly update transactions, and thus their number increases; this causes more updates to be propagated to the Backup[3], and thus it demonstrates a slight increase in its CPU utilization. The same effect occurs when a second Satellite is added to the system; the Master now executes even more updates, and thus more are propagated to the Hot Backup, and the higher its CPU utilization.

Finally, in order to visualize the effect of the configuration of the components of Multimed, we present the scalability that two different configurations achieve, in Figure 4.10. Table 4.4 presents the two different configurations demonstrated here.

The two system configurations differ in the number of cores assigned to the Master and the Dispatcher. In the case of the *Basic Configuration*, we over-provision both; in this case we can deploy up to 5 Satellites in the system. In the *Tight Configuration*, we reduce the number of cores of the Master and the Dispatcher, in order to see the difference in performance; we reduce each by 4 cores and the result is that we can add in total 6 Satellites in the system.

In Figure 4.10 we can see both the Browsing and the Shopping mix scalability results. The Basic configuration scales linearly with the number of Satellites added in the system for the Browsing mix, while in the Shopping mix, at the fifth Satellite we observe a degradation in the scalability. This degradation is due to

[3]The updates are propagated to the Satellites as well, but the utilization due to the updates received by the Master is not visible; for this reason we explain this only in the context of the Backup.

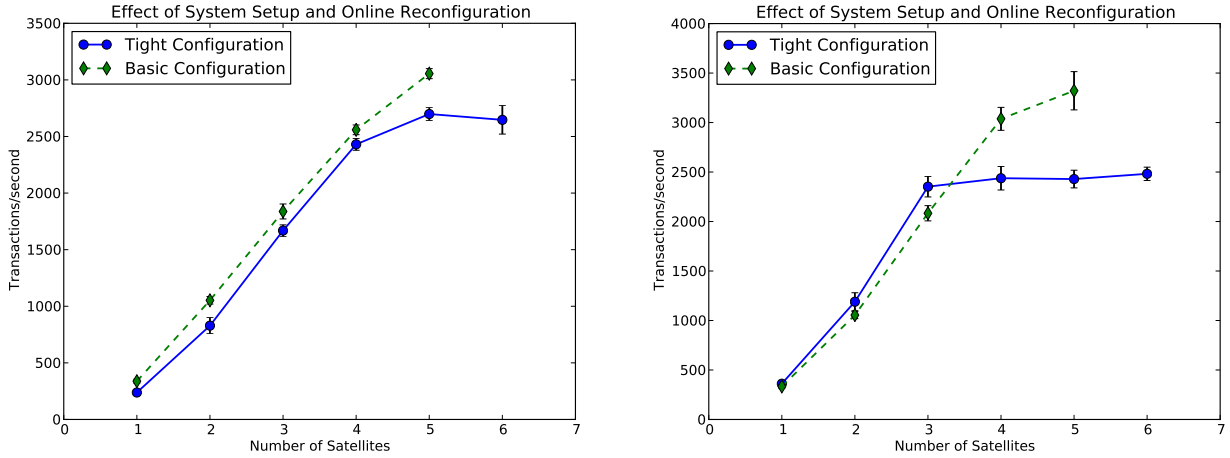
Component	CPUs
Master Node	16
Dispatcher	7
Hot Backup	1
Satellites	8

(a) Basic Configuration

Component	CPUs
Master Node	12
Dispatcher	3
Hot Backup	1
Satellites	8

(b) Tight Configuration

Table 4.4: The two different system configurations



(a) Browsing mix

(b) Shopping mix

Figure 4.10: An example of the difference that the configuration of the system can make

high CPU utilization in the Master node; remember that the Shopping mix has twice the rate of updates in the load. The Tight Configuration demonstrates degraded scalability earlier, for both mixes. In the Browsing mix, for more than 4 Satellites, the Dispatcher reaches very high CPU utilization levels, and for this reason the performance degrades; 3 cores only are not enough to handle all the computation and networking in the Dispatcher. In the case of the Shopping mix though, it is both the Dispatcher and the Master that become over-utilized; the Dispatcher for similar reasons as in the Browsing mix, while the Master, because of the high rate of updates in the load.

Evidently the Online Reconfiguration can be useful to tackle cases of misconfiguration and intolerance of the load; one can even extend the system with automatic procedures that monitor the system's performance, detect such performance degradations and take action by modifying the configuration of resources in the system.

Chapter 5

Discussion

During the design, implementation and evaluation of the system, we have at times encountered various interesting points to consider. In this section, we elaborate on the insight that our experience with Multimed has brought us.

5.1 Configuration and placement

Based on an already proven design, Multimed depends on its proper configuration in order to achieve high performance and scalability. The range of knobs starts from the placement of the various components of the system, until detailed tuning parameters of the database engine. The most important have been demonstrated and discussed in context, within this work.

During the experimentation with Multimed, we have followed heuristic guidelines in order to decide on the system's configuration and placement of components. An important decision to make when configuring Multimed, is how to allocate resources for each of the components. Trial and error has lead us to the optimal configurations, but this does not imply that the ones used during this work are the only ones that perform well. For the Browsing mix, for example, we have chosen Satellites with 4 cores in the evaluation of the horizontal scalability of the system, because, to our knowledge, this setup performs well. Having larger Satellites might have its benefits as well, and this is something to discover through experimentation.

When distributing Multimed on a cluster of multicores, we perceive the available multicores as a resource pool, and distribute resources accordingly. However this can be rather complicated, and there exist trade-offs related to node placement. Our system model takes the approach of placing the datasets of Satellites in main memory; this is a determinant factor though, when choosing how many Satellites to place within each multicore, since the main memory of the multicores is limited. The same decision has to be made for the cores of each component.

In the design of the system, there is nothing that dictates us to group components in multicores in one way or another. Typically a multicore can host more than one components, and the decision of which ones to group is made mainly on the resource allocation scheme that we follow. However, there is one factor that affects the decision of placement: the Domain-0. As seen in Section 4.3.3, the Dispatcher is a demanding component in terms of its networking, and networking uses communication channels that go through the Domain-0 of

the VMM hosting the Dispatcher. This lead us to the conclusion that, for cases where we intend to use high number of Satellites, or when very high throughput is expected, the Dispatcher should probably be placed on its own within a multicore, or at least with a component that will have reduced network traffic (such as the Hot Backup). This comes from the fact that the optimizations that we performed on Domain-0 in Section 4.1.5 are limited by the number of VIFs we can use (8) to split the traffic between the Domain-0 and the Dispatcher.

Moreover, when configuring the Satellites, one must take care in order to create equally-sized Satellites, meaning that the same number of cores and amount of memory should be given to each. The reason is that the Dispatcher of the system does not consider discrepancies in computational power in the Satellites; configuring unbalanced Satellites would result in unbalanced routing of the transactions. This can be seen as a future direction for Multimed, since it is not complicated to design other metrics on which the routing decisions will be made.

5.2 NUMA Awareness

In multicore architectures, the Non-Uniform Memory Access memory design is becoming increasingly important. Transactional data processing systems that run on multicores, like Multimed, ought to consider the NUMA configuration of the machine in order to achieve the best possible computation performance. A demonstration of this can be seen in the study of reader-writer locks in NUMA architectures [3].

Multimed allows for NUMA awareness; virtualization enables the system to set up specific core-set assignments to domains. Observing the NUMA configuration of the machine, one can easily set up assignments that respect the NUMA configuration, by selecting sets of cores that optimize the *distance* between NUMA *nodes*. A NUMA node consists of a block of memory, and the CPUs, I/O, etc. that are physically on the same bus as the memory [21]. The distance between the nodes, is a metric that can vary (hop count, latency, bandwidth), and typically is intended to show that the smaller the distance, the better the performance between the nodes, and to quantify this measure.

NUMA awareness mainly benefits the system by reducing the number of CPU cores competing for access on a shared memory bus. In multicores, this contention increases with the number of cores, and can possibly cause performance degradation. By using cores that belong to NUMA nodes with small distance in between, one achieves to speed up the access to the shared memory buses. Multimed assigns exclusive sets of cores to its components; this is in line with the NUMA configuration of the system, since less cores means less contention on the shared memory buses, and thus less synchronization overhead. For this reason, the components of Multimed are assigned cores that reside on the same (or, if not possible, on the closest) NUMA node.

5.3 Future Directions

The achievements of the system so far, lead to several possible directions for future work; in fact, there is already ongoing work towards these directions. Firstly, by building in elasticity to the system, we gain new prospects by adding automatic adaptation to changes in the load. The reconfiguration operations that we have extended Multimed with, can be leveraged automatically at runtime in order to adapt the system

to characteristics of the load or target functions. For example, by monitoring the system's performance, in terms of throughput and response time, one can create thresholds of performance and take specific actions once they are reached. Setting an upper bound to the response time of the system, will result in constantly taking action (adding resources to the system) until the response time drops below the acceptable bounds. Additionally, one can enforce thresholds on the performance of individual components; once the CPU utilization of the Satellites reaches a high level, the system can automatically add a new Satellite. Similarly, once the cores of the Master or Dispatcher of the system are fully utilized, one could add more cores to these components. The automatic reconfiguration of the system might require more involved monitoring procedures. For instance, Node-level reconfiguration of the system is very fast and can even be utilized in cases of load bursts; however, System-level reconfiguration is not as fast, and its completion time depends on the dataset size. The decision to add a new Satellite to the system must be thus based on longer monitoring of the system, in order to ensure that the load has increased globally and not for a short period of time.

Another argument is the limitation of the system's scalability because of the proportion of reads and writes in the load. A single Master node can only sustain a certain amount of write load; even using fast disks, their limit will always become present before the limit of CPU. Within the scope of Multimed, one could perform Master Node distribution as well. By replicating the Master into multiple instances, the write load could be also separated in the same way that the read load is now separated among the Satellites. The challenge then is to build mechanisms for maintaining consistency of the dataset among the multiple Masters; this is possible though, with acceptable overhead, as shown in [19], using coordination and consistency techniques of distributed computing. The vision in this case goes even further; Multimed could then form super-clusters of Masters (that still replicate the same dataset), while each of them forms a cluster of Satellites in the present form of the system.

The approach of multiple Masters in the system might lead to other directions as well: multi-tenancy. In DBFarm [14], multiple instances of databases are deployed and managed within the same system. This is a very interesting prospect for Multimed as well, and would enable the system of providing its already established scalability benefits for numerous different database instances and thus applications. This direction is part of the ongoing work on Multimed.

5.4 TPCE

During the work of this thesis, there has been an effort for a Java implementation of the TPC-E Benchmark[24] of the Transaction Processing Performance Council. TPC-E simulates the OTLP workload of a brokerage firm, that executes transactions on the firm's customer accounts on a central database server. This benchmark focuses on a variety of system components, by generating load that, among others, includes the following characteristics:

- Diverse transaction types, with diverse complexities, being executed concurrently.
- Moderate system and application execution time.
- A mixture of disk I/O and processor usage.
- ACID properties.

More precisely, the benchmark consists of 12 distinct transactions, 4 of which are read-write transactions, and add up to the 32.1% of the total load. The transactions pertain to customer initiated operations, or market related operations; the market related operations are simulated in the Market Exchange Emulator (MEE). The MEE is a component of the benchmark that differentiates its execution from the rest of the transactions; it simulates the creation and execution of trades in the financial market. The MEE is implemented as a series of separate threads that generate load to the system; up to 20% of the total read-write transactions originate from the MEE threads.

The implementation of TPC-E was complicated and time consuming; for this reason we have been unable to fully test our system with TPC-E. Our experience with it, in the context of Multimed, shows so far that TPC-E puts significant pressure to the Master Node and its transaction log. We have encountered high utilization of the log (up to 60%) even for small loads (generated by 50 clients), and we have been unable to devise a configuration of the system that would allow TPC-E generated load to scale linearly with the resources in Multimed. For the sake of reference, we have tried several approaches for the placement of the transaction log; a standard disk, an SSD, and two SSDs forming a RAID-0 array. The aforementioned utilization values regard the case of the RAID-0 array. In order to be able to comprehend TPC-E and make conclusions on how it can be used on Multimed, we have to delve deeper into the aspects of the load, like the MEE threads, their effect on the transaction log, possible optimizations for the log and the Master, and so on.

In the context of the results of Section 4.3, since even Shopping turned out to saturate our system's performance quickly, we expect to encounter similar behaviour with TPC-E. However, this is only the observation that brief experimentation has allowed us for; before we extensively analyze the behaviour of TPC-E and the effect of the relevant configuration aspects, we cannot characterize further the usage of TPC-E on Multimed.

Chapter 6

Conclusion

With this thesis we have explored significant properties of transactional data processing systems in the context of modern multicore architectures. Scalability has been studied thoroughly in the past, but the modern multicore architectures challenge it further. Multimed is a system that achieves linear scalability with the number of cores in a multicore machine, and outperforms its stand alone database engine equivalents, in most cases. The very design of the system has enabled us to extend Multimed to a cluster of multicores, instead of a single multicore machine; thus we have achieved linear horizontal scalability for the system, with the number of multicores in its cluster.

At the same time, we have introduced elasticity in Multimed. Modern large scale systems ought to be able to adapt to the demand of the load by rearranging and reconfiguring the available resources. We have achieved this through virtualization; a proven and readily available technology that follows the trends in cloud architectures. Virtualization has enabled us to form an abstraction on the resources offered by the cluster of multicores, into a uniform pool, common among the components of the system. At the same time, its intriguing virtual machine dynamic reconfiguration capabilities have provided a matching reconfiguration mechanism to Multimed, according to the specifications that we envision. Finally, a method for dynamic addition of Satellites to the system completed the Online Reconfiguration mechanism of Multimed.

An evaluation of the system has demonstrated the performance and scalability gains that our approach achieves, as compared to the limits of a stand alone database engine. The approach of Multimed for separating the load in reads and writes, and also distributing the load among numerous replicated databases, was successful in achieving both vertical and horizontal scalability. In parallel, the demonstration of the performance of the system has provided with insightful reasoning regarding Online Reconfiguration. Finally, the evaluation of the system allowed us to gain a deeper understanding of virtualization; we have studied its overhead in our system, and realized that despite the overhead, the abstraction and reconfiguration options that it brings to our system, compensate for it, resulting in a uniform and concrete system design.

The evaluation of the system reveals its limitations as well. The scalability of the system is limited mainly by the Master Node's capability of executing write load, and this depends on the characteristics of its hardware. To the extend that the CPU of the Master limits performance, Multimed provides the solution through reconfiguration. However, the persistent storage of the Master can impose limitations out of the reach of Multimed. Write-intensive loads are thus less suitable for Multimed, and indeed for any primary copy replication approach.

Conclusively, Multimed proposes a system architecture that enables transactional data processing scalability in multicore machines, and in clusters of multicores, without having to perform any modifications to existing database engines, while achieving significant performance benefits, as compared to conventional databases. The scalability and elasticity features of Multimed, bring new potential to the system in exploring multi-tenancy and further distribution, automatic reconfiguration, and other directions that remain to be seen.

Bibliography

[1] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore. Database scalability, elasticity, and autonomy in the cloud - (extended abstract). In *DASFAA (1)*, pages 2–15, 2011.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[3] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *PPOPP*, pages 157–166, 2013.

[4] E. Cecchet. C-jdbc: a middleware framework for database clustering. *IEEE Data Eng. Bull.*, 27(2):19–26, 2004.

[5] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[6] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[7] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[8] S. Harizopoulos and A. Ailamaki. Stageddb: Designing database servers for modern hardware. *IEEE Data Eng. Bull.*, 28(2):11–16, 2005.

[9] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In *DaMoN*, pages 21–26, 2009.

[10] R. Johnson, I. Pandis, and A. Ailamaki. Improving oltp scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[11] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[12] V. Pankratius and M. Heneka. Moving database systems to multicore: An auto-tuning approach. In *ICPP*, pages 582–591, 2011.

[13] K. Papadopoulos, K. Stavrou, and P. Trancoso. Helpercoredb: Exploiting multicore technology to improve database performance. In *IPDPS*, pages 1–11, 2008.

[14] C. Plattner, G. Alonso, and M. T. Özsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware*, pages 180–200, 2006.

[15] C. Plattner, G. Alonso, and M. T. Özsu. Extending dbmss with satellite databases. *VLDB J.*, 17(4):657–682, 2008.

[16] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *EuroSys*, 2013.

[17] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *EuroSys*, pages 17–30, 2011.

[18] M. Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

[19] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433, 2005.

[20] The irqbalance utility. `https://code.google.com/p/irqbalance/`.

[21] Numa architecture faqs. `http://lse.sourceforge.net/numa/faq/#what_is_distance`.

[22] Postgresql version 9.2 press release. `http://www.postgresql.org/about/press/presskit92/`.

[23] Postgresql write ahead log. `http://www.postgresql.org/docs/9.2/static/wal-intro.html`.

[24] The tpc-e benchmark specification. `http://www.tpc.org/tpce/`.

[25] The tpc-w benchmark specification. `http://www.tpc.org/tpcw/`.

[26] Xen architecture overview. `http://wiki.xen.org/wiki/Xen_Overview`.

[27] Xen paravirtualization. `http://wiki.xen.org/wiki/Paravirtualization_(PV)`.

[28] Xen virtualization spectrum. `http://wiki.xen.org/wiki/Virtualization_Spectrum`.

List of Figures

List of Tables