Database Engines on Multicores, Why Parallelize When You Can Distribute? *

Tudor-Ioan Salomie Ionut Emanuel Subasu Jana Giceva Gustavo Alonso

Systems Group, Computer Science Department
ETH Zurich, Switzerland
{tsalomie, subasu, gicevaj, alonso}@inf.ethz.ch

Abstract

Multicore computers pose a substantial challenge to infrastructure software such as operating systems or databases. Such software typically evolves slower than the underlying hardware, and with multicore it faces structural limitations that can be solved only with radical architectural changes. In this paper we argue that, as has been suggested for operating systems, databases could treat multicore architectures as a distributed system rather than trying to hide the parallel nature of the hardware. We first analyze the limitations of database engines when running on multicores using MySQL and PostgreSQL as examples. We then show how to deploy several replicated engines within a single multicore machine to achieve better scalability and stability than a single database engine operating on all cores. The resulting system offers a low overhead alternative to having to redesign the database engine while providing significant performance gains for an important class of workloads.

Categories and Subject Descriptors H.2.4 [*Information Systems*]: DATABASE MANAGEMENT—Systems

General Terms Design, Measurement, Performance

Keywords Multicores, Replication, Snapshot Isolation

1. Introduction

Multicore architectures pose a significant challenge to existing infrastructure software such as operating systems [Baumann 2009; Bryant 2003; Wickizer 2008], web servers [Veal 2007], or database engines [Hardavellas 2007; Papadopoulos 2008].

In the case of relational database engines, and in spite of the intense research in the area, there are still few practical solutions that allow a more flexible deployment of databases over multicore machines. We argue that this is the result of the radical architectural changes that many current proposals imply. As an alternative, in this paper we describe a solution that works well in a wide range of use cases and requires no changes to the database engine. Our approach is intended neither as a universal solution to all use cases nor as a replacement to a much needed complete redesign of the database engine. Rather, it presents a new architecture for database systems in the context of multicores relying on well known distributed system techniques and proving to be widely applicable for many workloads.

1.1 Background and Trends

Most commercial relational database engines are based on a decades old design optimized for disk I/O bottlenecks and meant to run on single CPU computers. Concurrency is achieved through threads and/or processes with few of them actually running simultaneously. Queries are optimized and executed independently of each other with synchronization enforced through locking of shared data structures. All these features make the transition to modern hardware platforms difficult.

It is now widely accepted that modern hardware, be it multicore, or many other developments such as flash storage or the memory-CPU gap, create problems for current database engine designs. For instance, locking has been shown to be a major deterrent for scalability with the number of cores [Johnson 2009a] and the interaction between concurrent queries when updates or whole table scans are involved can have a severe impact on overall performance [Unterbrunner 2009]. As a result, a great deal of work proposed either ways to modify the engine or to completely redesign the architecture. Just to mention a few examples, there are proposals to replace existing engines with pure main memory scans [Unterbrunner 2009]; to use dynamic

* This work is partly funded by the Swiss National Science Foundation under the programs ProDoc Enterprise Computing and NCCR MICS

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

programming optimizations to increase the degree of parallelism for query processing [Han 2009]; to use *helper cores* to efficiently pre-fetch data needed by working threads [Papadopoulos 2008]; to modularize the engine into a sequence of stages, obtaining a set of self-contained modules, which improve data locality and reduce cache problems [Harizopoulos 2005]; or to remove locking contention from the storage engine [Johnson 2009a,b]. Commercially, the first engines that represent a radical departure from the established architecture are starting to appear in niche markets. This trend can be best seen in the several database appliances that have become available (e.g., TwinFin of IBM/Netezza, and SAP Business Datawarehouse Accelerator; see [Alonso 2011] for a short overview).

1.2 Results

Inspired by recent work in *multikernel* operating systems [Baumann 2009; Liu 2009; Nightingale 2009; Wentzlaff 2009], our approach deploys a database on a multicore machine as a collection of distributed replicas coordinated through a middleware layer that manages consistency, load balancing, and query routing. In other words, rather than redesigning the engine, we partition the multicore machine and allocate an unmodified database engine to each partition, treating the whole as a distributed database.

The resulting system, *Multimed*, is based on techniques used in LANs as part of computer clusters in the Ganymed system [Plattner 2004], adapted to run on multicore machines. Multimed uses a primary copy approach (the *master database*) running on a subset of the cores. The master database receives all the update load and asynchronously propagates the changes to *satellite databases*. The satellites store copies of the database and run on non overlapping subsets of the cores. These satellites are kept in sync with the master copy (with some latency) and are used to execute the read only load (queries). The system guarantees global consistency in the form of snapshot isolation, although alternative consistency guarantees are possible.

Our experiments show that a minimally optimized version of Multimed exhibits both higher throughput with lower response time and more stable behavior as the number of cores increase than standalone versions of PostgreSQL and MySQL for standard benchmark loads (TPC-W).

1.3 Contribution

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

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

Finally, in this paper we show a few simple optimizations to improve the performance of Multimed. For instance, partial replication is used to reduce the memory footprint of the whole system. Many additional optimizations are possible over the basic design, including optimizations that do not require modification of the engine, (e.g., data placement strategies, and specialization of the satellites through the creation of indexes and data layouts tailored to given queries).

The paper is organized as follows. In the next section we motivate Multimed by analyzing the behavior of PostgreSQL and MySQL when running on multicore machines. The architecture and design of Multimed are covered in section 3, while section 4 discusses in detail our experimental evaluation of Multimed. Section 6 discusses related work and section 7 concludes the paper.

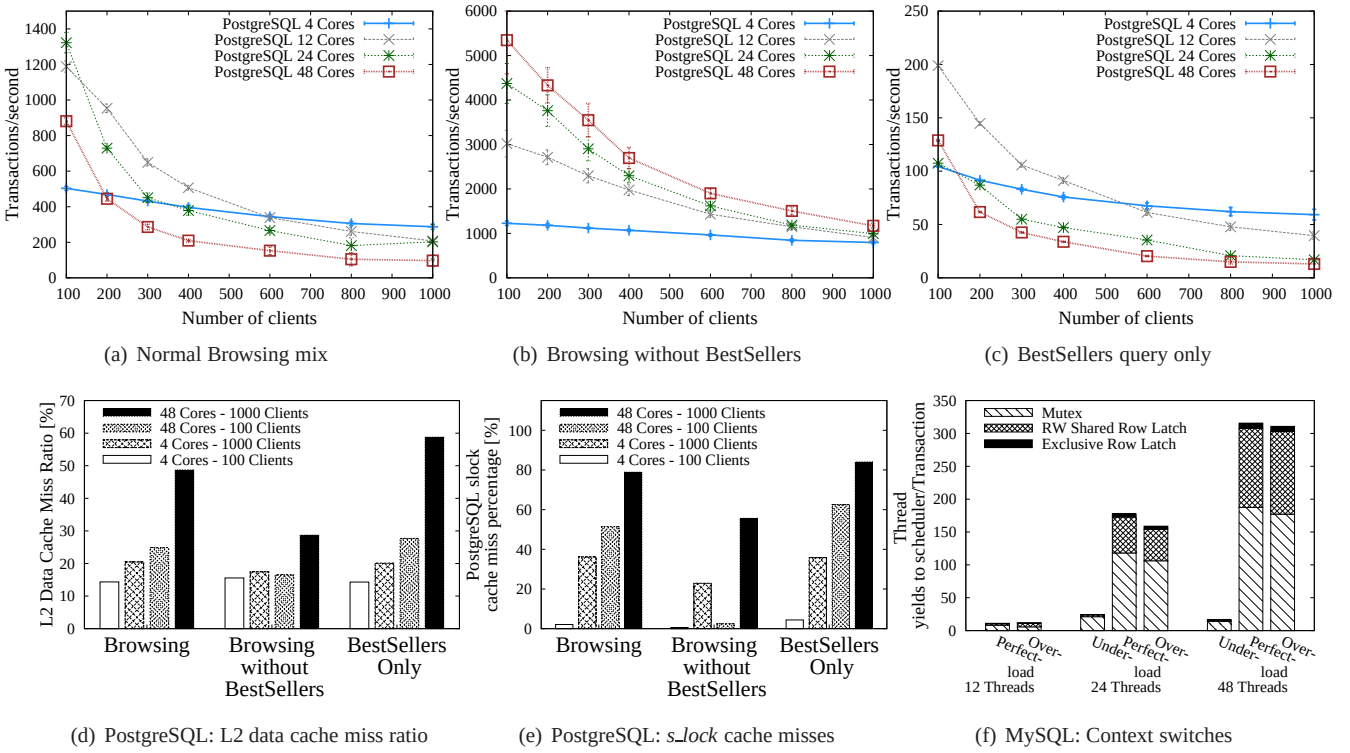
2. Motivation

To explore the behavior of traditional architectures in more detail, we have performed extensive benchmarks over PostgreSQL and MySQL (open source databases that we can easily instrument and where we can map bottlenecks to concrete code sequences). Our analysis complements and confirms the results of similar studies done on other database engines over a variety of multicore machines [Hardavellas 2007; Johnson 2009b; Papadopoulos 2008].

The hardware configuration and database settings used for running the following experiments are described in section 4.1. The values for L2 cache misses and context switches were measured using a runtime system profiler [OProfile].

2.1 Load interaction

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



(a) Normal Browsing mix (b) Browsing without BestSellers (c) BestSellers query only

(d) PostgreSQL: L2 data cache miss ratio (e) PostgreSQL: *s_lock* cache misses (f) MySQL: Context switches

Figure 1. Current databases on multicore

We have investigated load interaction in both PostgreSQL and MySQL using the *Browsing* mix of the TPC-W Benchmark (see below for details on the experimental setup).

PostgreSQL's behavior with varying number of clients and cores is shown for the *Browsing* mix in figure 1(a); for all other queries in the mix except *BestSellers* in figure 1(b); and for the *BestSellers* query only in figure 1(c).

For the complete mix (figure 1(a)), we observe a clear performance degradation with the number of cores. We traced the problem to the *BestSellers* query, an analytical query that is performing scans and aggregation functions over the three biggest tables in the database. On one hand the query locks a large amount of resources and, while doing this, causes a large amount of context switches. On the other hand all the concurrent queries have to wait until the *BestSellers* query releases the locked resources. When this query is removed from the mix, figure 1(b), the throughput increases by almost five times and now it actually improves with the number of cores. When running the *BestSellers* query alone (figure 1(c)), we see a low throughput due to the interference among concurrently running queries and, again, low performance as the number of cores increases.

The interesting aspect of this experiment is that *BestSellers* is a query and, as such, is not doing any updates. The negative load interaction it causes arises from the competition for resources, which becomes worse as the larger number of cores allows us to start more queries concurrently. Similar effects have been observed in MySQL, albeit for loads involving full table scans [Unterbrunner 2009]. Full table scans require a lot of memory bandwidth and slow down any other concurrent operation, providing another example of negative load interaction that becomes worse as the number of cores increases.

2.2 Contention

One of the reasons why loads interact with each other is contention. Contention in databases is caused mainly by concurrent access to locks and synchronization primitives.

To analyze this effect in more detail, we have profiled PostgreSQL while running the *BestSellers* query. The individual run time for this query, running alone in the system, is on average less than 80ms, indicating that there are no limitations in terms of indexing and data organization.

Figure 1(d) shows the L2 data cache misses for the full Browsing mix, the Browsing mix without the *BestSellers* and the *BestSellers* query alone. The L2 data cache miss ratio was computed using the expression below based on measured values for L2 cache misses, L2 cache fills and L2 requests (using the CPU performance counters). We have done individual measurements for each CPU core, but as there are no significant differences between the cores, we used the averaged values of the measured metrics.

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

With more clients and cores, we see a high increase in cache misses for the workloads containing the *BestSellers* query. We have traced this behavior to the "s_lock" (spin lock) function, which is used in PostgreSQL to control access to the shared buffers data structures (held in shared memory). Every time a lock can not be acquired, a context switch takes place, forcing an update of the L2 cache.

Figure 1(e) shows that the time spent on the "s_lock" function increases with both clients and cores, only when the *BestSellers* query is involved. We would expect to see an increase with the number of clients but not with more cores. Removing again the *BestSellers* query from the mix, we observe that it is indeed the one that causes PostgreSQL to waste CPU cycles on the "s_lock" function as the number of cores increases. Finally, looking at the "s_lock" while running only the *BestSellers* query we see that it dictates the behavior of the entire mix.

The conclusion from these experiments is that, as the number of cores and clients increase, the contention on the shared buffers significantly degrades performance: more memory leads to more data under contention, more cores just increase the contention. This problem that has also been observed by Boyd-Wickizer [2010].

The performance of MySQL for the *Browsing* mix with different number of cores and clients is shown in figure 5. MySQL's InnoDB storage engine acts as a queuing system: it has a fixed number of threads that process client requests (storage engine threads). If more client requests arrive than available threads, MySQL will buffer them until the previous ones have been answered. In this way MySQL is not affected by the number of clients but it shows the same pathological behavior as PostgreSQL with the number of cores: more cores result in lower throughput and higher response times.

While running this experiment, we monitored the times a thread had to yield to the OS due to waits for a lock or a latch. Figure 1(f) presents the number of thread yields per transaction for different loads on the system.

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

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

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

2.3 Our approach

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

Unlike existing work that focuses on optimizing the access time to shared data structures [Hardavellas 2007; Johnson 2009b] or aims at a complete redesign of the engine [Harizopoulos 2005; Unterbrunner 2009], Multimed does not require code modifications on the database engine. Instead, we use replicated engines each one of them running on a non-overlapping subset of the cores.

3. The Multimed System

Multimed is a platform for running replicated database engines on multicore machines. It is independent of the database engine used, its main component being a middleware layer that coordinates the execution of transactions across the replicated database engines. The main roles of Multimed are: (i) mapping database engines to hardware resources, (ii) scheduling and routing transactions over the replicated engines and (iii) communicating with the client applications.

3.1 Architectural overview

From the outside, Multimed follows the conventional client-server architecture of database engines. Multimed's client component is implemented as a JDBC Type 3 Driver. Internally, Multimed (figure 2) implements a master-slave replication strategy but offers a single system image, i.e., the clients see a single consistent system. The master holds a primary copy of the data and is responsible for executing all updates. Queries (the read only part of the load) run on the satellites. The satellites are kept up to date by asynchronously propagating *WriteSets* from the master. To provide a consistent view, queries can be scheduled to run on a satellite node only after that satellite has done all the updates executed by the master prior to the beginning of the query.

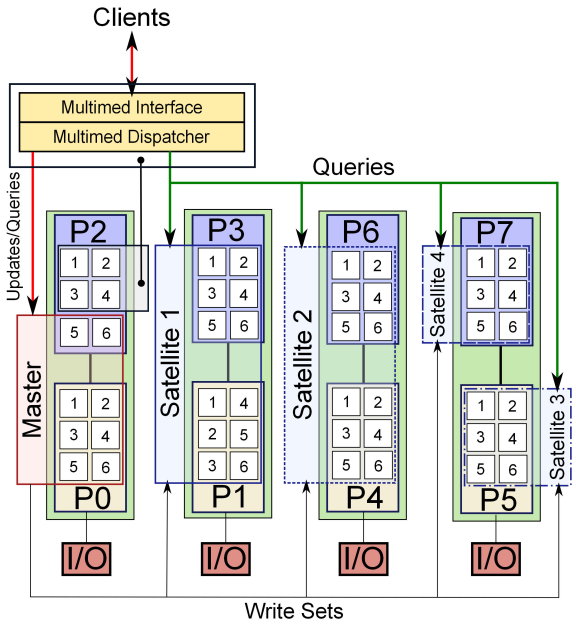


Figure 2. A possible deployment of Multimed

The main components of Multimed are the *Communication* component, the *Dispatcher* and the *Computational Nodes*. The Communication component implements an asynchronous server that allows Multimed to process a high number of concurrent requests. Upon receiving a transaction, Multimed routes the transaction to one of its *Computational Nodes*, each of which coordinates a database engine. The routing decision is taken by the *Dispatcher* subsystem.

With this architecture, Multimed achieves several goals. First, updates do not interfere with read operations as the updates are executed in the master and reads on the satellites; second, the read-only load can be separated across replicas so as to minimize the interference of heavy queries with the rest of the workload.

3.2 How Multimed works

We now briefly describe how Multimed implements replication, which is done by adapting techniques of middleware based replication [Plattner 2004] to run in a multicore machine. In a later section we explore the optimizations that are possible in this context and are not available in network based systems.

3.2.1 Replication model

Multimed uses lazy replication [Gray 1996] between its master and satellite nodes but guarantees a consistent view to the clients. The master node is responsible for keeping a durable copy of the database which is guaranteed to hold the latest version of the data. All the update transactions are executed on the master node as well as any operation requiring special features such as stored procedures, triggers, or user defined functions.

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

Each time an update transaction is committed, the master commits the transaction locally. The master propagates changes as a list of rows that have been modified. A satellite enqueues these update messages and applies them in the same order as they were executed at the master node.

Multimed enforces snapshot isolation as a consistency criterion (see [Daudjee 2006] for a description of snapshot isolation and other consistency options such as session consistency in this type of system). In snapshot isolation queries are guaranteed to see all changes that have been committed at the time the transaction started, a form of multiversion concurrency control found today in database engines such as Oracle, SQLServer, or PostgreSQL. When a query enters the system, the *Dispatcher* needs to decide where it can run the transaction (i.e., to which node it should bind it, which may involve some small delay until a copy has all necessary updates) and if multiple options are available, which one to choose. Note that the master node is always capable of running any query without any delay and can be used as a way to minimize latency if that is an issue for particular queries.

Within each database, we rely on the snapshot isolation consistency of the underlying database engine. This means that an update transaction will not interfere with read transactions, namely the update transaction is applied on a different "snapshot" of the database and once it is committed, the shadow version of the data is applied on the active one. In this way, Multimed can schedule queries on replicas at the same time they are being updated.

3.2.2 WriteSet extraction and propagation

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

```
input : Connection con, Server Count Number scn
WriteSet ws ← con.getWriteSet();
if ws.getStatementCount() > 0 then
    synchronized lock_object
        con.commit();
        ws.setSCN(scn.atomicIncrement());
        foreach satellite sat do sat.enqueue(ws);
    end
else con.commit();
```

Algorithm 1: WriteSet Propagation

When an update transaction is committed on the master node, the WriteSet is extracted (by invoking a server side function), parsed and passed on to each satellite for enqueuing in its update queue, following algorithm 1.

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

3.3 Multimed's components

The main components of Multimed are the computational nodes, the dispatcher, the system model and the client communication interface.

3.3.1 Computational nodes

A *Computational Node* is an abstraction over a set of hardware (CPUs, memory, etc.) and software (database engine and stored data, connection pools, queues, etc) resources. Physically, it is responsible for forwarding queries and updates to its database engine.

Each *Computational Node* runs in its own thread. It is in charge of (i) executing queries; (ii) executing updates and (iii) returning results to the clients. Each transaction is bound to a single *Computational Node* (can be the master) which has the capability of handling all the requests that arrive in the context of a transaction.

Multimed has one master *Computational Node* and any number of satellite *Computational Nodes*. Upon arrival, queries are assigned a timestamp and dispatched to the first satellite available that has all the updates committed up to that timestamp (thereby enforcing snapshot isolation). Satellite nodes do not need to have any durability guarantees (i.e., do not need to write changes to disk). In the case of failure of a satellite, no data is lost, as everything is durably committed by the master *Computational Node*.

3.3.2 Dispatcher

The Multimed *Dispatcher* binds transactions to a Computational Node. It routes update transactions to the master node, leaving the read transactions to the satellites. The differentiation of the two types of transactions can be done based on the transaction's *readOnly* property (from the JDBC API).

The Dispatcher balances load by choosing the most lightly loaded satellite from among those that are able to handle the transaction. The ability of a satellite to handle a transaction is given by the freshness of the data it holds and by the actual data present (in the case of partial replication). The load can be the number of active transactions, the CPU usage, average run time on this node, etc. When no capable satellite is found, the *Dispatcher* waits until it can bind to a satellite with the correct update level or it may choose to bind the transaction to the master node.

3.3.3 System model

The system model describes the configuration of all *Computational Nodes*. It is used by the *Dispatcher* when processing client requests.

The *System Model* currently defines a static partitioning of the underlying software and hardware resources (dynamic partitioning is left for future work as it might involve reconfiguring the underlying database). It is used at start-time to obtain a logical and physical description of all the *Computational Nodes* and of the required connection settings to the underlying databases. It also describes the replication scheme in use, specifying what data is replicated where, the tables that are replicated, and where transactions need to be run.

3.3.4 Communication component

The communication subsystem, on the server side, has been implemented based on *Apache Mina 2.0* [Apache Mina]. The communication interface is implemented as an asynchronous server using *Java NIO* libraries. Upon arrival, each client request is passed on to a *Computational Node* thread for processing. We have implemented the server component of the system as a non-blocking message processing system so as to be able to support more concurrent client connections than existing engines. This is important to take advantage of the potential scalability of Multimed as often the management of client connections is a bottleneck in database engines (see the results for PostgreSQL above).

3.4 System optimizations

Multimed can be configured in many different ways and accepts a wide range of optimizations. In this paper we describe a selection to illustrate how Multimed can take advantage of multicore systems in ways that are not possible in conventional engines.

On the communication side, the messages received by the server component from the JDBC Type 3 Driver are small (under 100 bytes). By default, multiple messages will

be packed together before being sent (based on Nagle's algorithm [Peterson 2000]), increasing the response time of a request. We disabled this by setting the TCP_NODELAY option on the Java sockets, reducing the RTT for messages by a factor of 10 at the cost of a higher number of packets on the network.

On the server side, all connections from the *Computational Nodes* to the database engines are done through JDBC Type 4 Drivers (native protocols) to ensure the best performance. Using our own connection pool increases performance as no wait times are incurred for creating/freeing a database connection.

At the *Dispatcher* level, the binding between an external client connection and an internal database connection is kept for as long as possible. This binding changes only when the JDBC *readOnly* property of the connection is modified.

For the underlying satellite node database engines, we can perform database engine-specific tweaks. For instance, for the PostgreSQL satellites, we turned off the synchronous commit of transactions and increased the time until these reach the disk. Consequently, the PostgreSQL specific options like *fsync*, *full_page_writes* and *synchronous_commit* were set to *off*, the *commit_delay* was set to its maximum limit of $100,000\mu$s, and the *wal_writer_delay* was set to $10,000$ms. Turning off the synchronous commit of the satellites does not affect the system, since they are not required for durability. Similar optimizations can be done with MySQL although in the experiments we only include the delay writes option. In the experimental section, we consider three optimization levels:

C0 implements full data replication on disk for all satellites. This is the naïve approach, where we expect performance gains from the reduced contention on the database's synchronization primitives, but also higher disk contention.

C1 implements full data replication in main memory for all satellites, thereby reducing the disk contention.

C2 implements partial or full data replication in main memory for the satellites and transaction routing at the *Dispatcher*. This approach uses far less memory than $C1$, but requires a-priori knowledge of the workload to partition the data adequately (satellites will be specialized for running only given queries). For the case of the 20GB database used in our experiments, a specialized replica containing just the tables needed to run the *BestSellers* query needs only 5.6GB thus allowing us to increase the number of in-memory satellite nodes. For CPU-bound use cases this approach allows us to easily scale to a large number of satellite nodes, and effectively push the bottleneck to the maximum disk I/O that the master database can use.

4. Evaluation

In this section we compare Multimed with conventional database engines running on multicore. We measure the throughput and response time of each system while running on a different number of cores, clients, and different database sizes. We also characterize the overhead and applicability of Multimed under different workloads. Aiming at a fair comparison between a traditional DBMS and Multimed, we used the TPC-W benchmark, which allows us to quantify the behavior under different update loads.

4.1 Setup

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

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

For the experiments with three and five satellites, each satellite was allocated entirely within a CPU, respectively within a die, to avoid competition for the cache. In the experiments with ten satellites, partial replication was used, making the databases smaller. In this case, each satellite was allocated on four cores for a total of 3 satellites per socket. Two of these satellites are entirely within a die and the third spawns two dies within the same CPU. Due to the small size of the replicas (the point we want to make with partial replication), we have not encountered cache competition problems when satellites share the L3 cache.

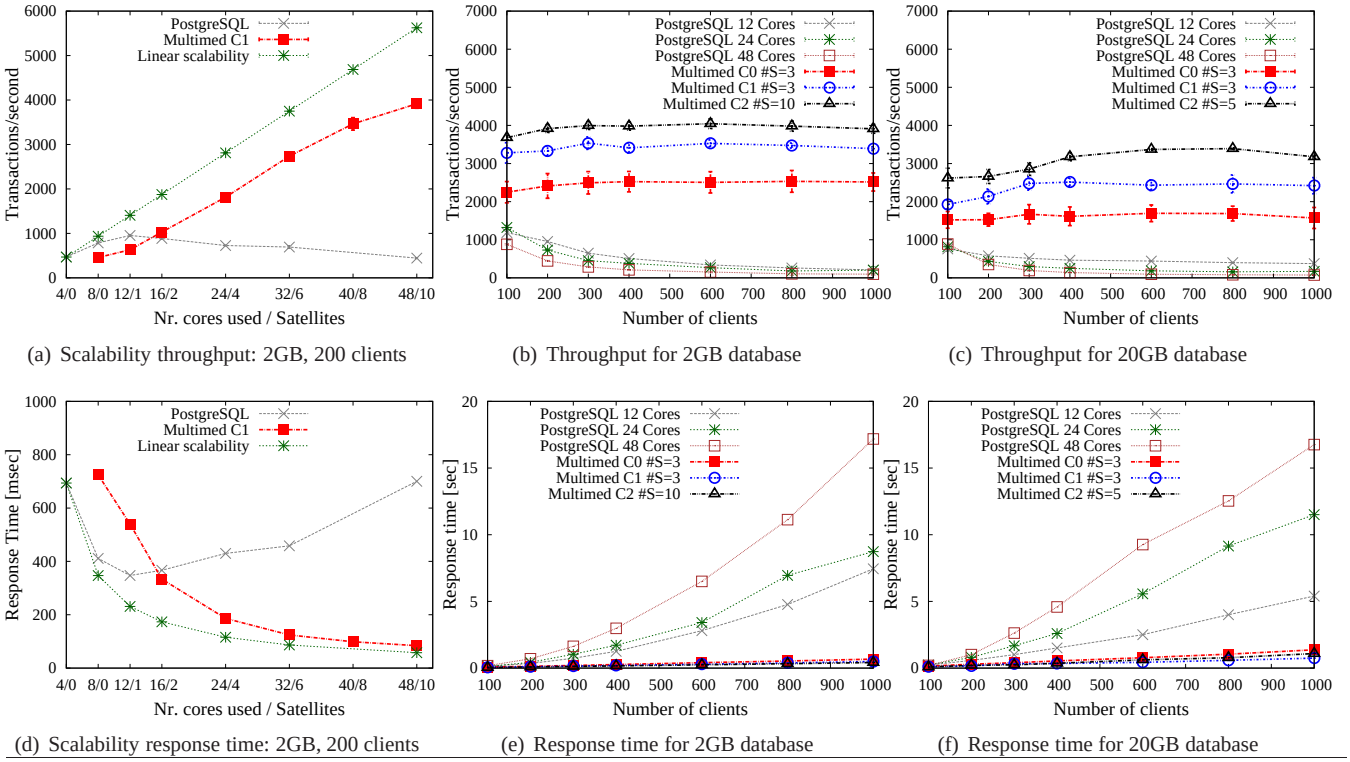
The hard disks in our machine prevented us from exploring more write intensive loads. In practice, network attached storage should be used, thereby allowing Multimed to support workloads with more updates. Nevertheless, the features and behavior of Multimed can be well studied in this hardware platform. A faster disk would only change at which point the the master hits the I/O bottleneck, improving the performance of Multimed even further.

The operating system used is a 64-bit Ubuntu 10.04 LTS Server, running PostgreSQL 8.3.7, MySQL 5.1 and Sun Java SDK 1.6.

4.2 Benchmark

The workload used is the TPC-W Benchmark over datasets of 2GB and 20GB. Each run consists of having the clients connect to the database and issue queries and updates, as per the specifications of the TPC-W mix being run. Clients issue queries for a time period of 30 minutes, without think times. Each experiment runs on a fresh copy of the database, so that dataset evolution does not affect the measurements. For consistent results, the memory and threading parameters of PostgreSQL and MySQL are fixed to the same values for both the standalone and Multimed systems.

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



(a) Scalability throughput: 2GB, 200 clients (b) Throughput for 2GB database (c) Throughput for 20GB database

(d) Scalability response time: 2GB, 200 clients (e) Response time for 2GB database (f) Response time for 20GB database

Figure 3. PostgreSQL: Standalone vs. Multimed, *Browsing* mix

emit the workload as well as to measure the throughput and response time.

The TPC-W benchmark specifies three workload mixes: TPC-W *Browsing* (10% updates), TPC-W *Shopping* (20% updates) and TPC-W *Ordering* (50% updates). Out of these three, we focus on the Browsing and Shopping mixes. The Ordering mix is disk intensive and hits an I/O bottleneck before any proper CPU usage is seen.

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

We have done extensive tests on Multimed, trying to find the optimal configuration to use in the experiments. The number of cores on which the satellites and the master nodes are deployed can be adjusted. Also, the number of cores allocated for Multimed's middleware code can be configured. In the experiments below we mention the number of satellites (#S) and the optimization (C0-C2) that were used.

4.3 PostgreSQL: Standalone vs. Multimed version

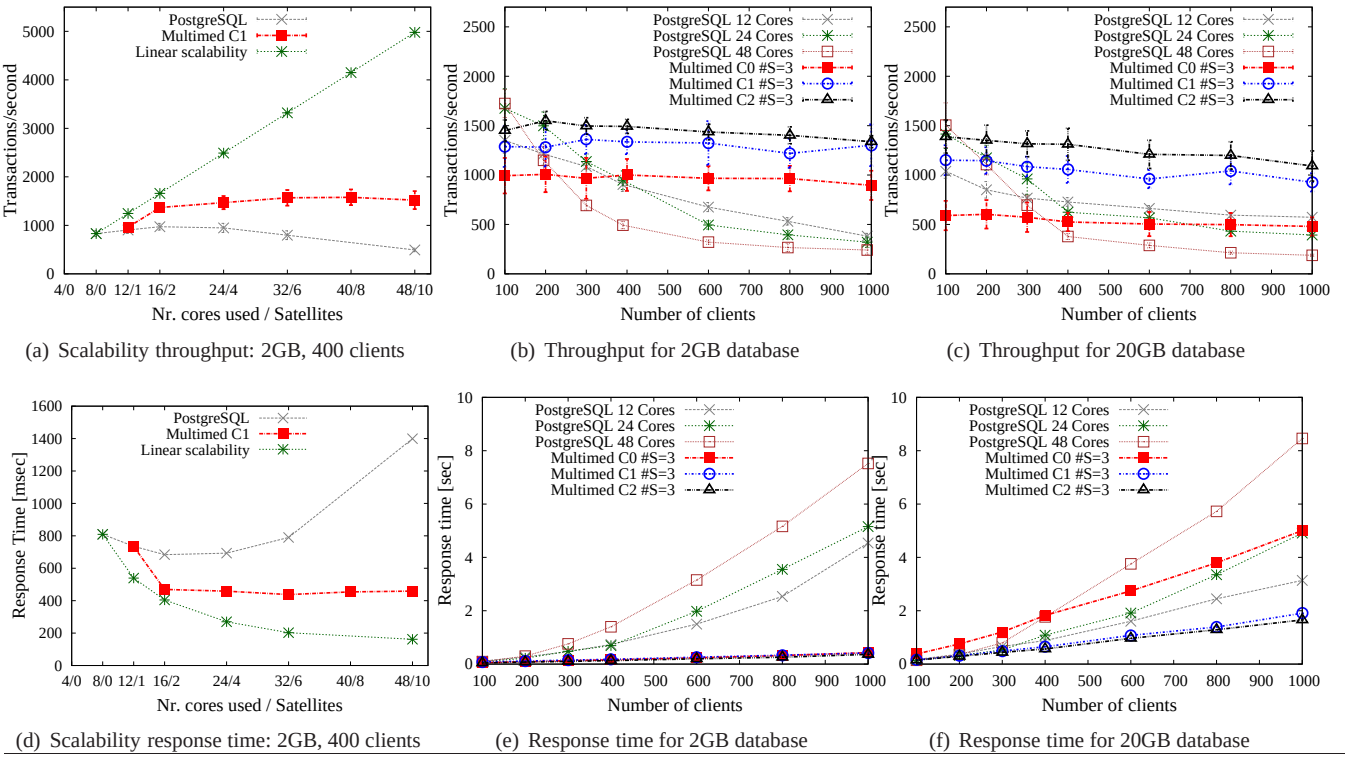
This section compares the performance of PostgreSQL and Multimed running on top of PostgreSQL.

4.3.1 Query intensive workload

Figures 3(a) and 3(d) present the scalability of PostgreSQL compared to Multimed $C1$, in the case of the 2GB database, and 200 clients. The x-axis shows the number of cores used by both Multimed and PostgreSQL, as well as the number of satellites coordinated by Multimed. Both the throughput (figure 3(a)) and the response time (figure 3(d)) show that the TPC-W Browsing mix places a lot of pressure on standalone PostgreSQL, causing severe scalability problems with the number of cores. Multimed running on 4 cores, the master node on 4 cores, and each satellite on 4 cores scales up almost linearly to a total of 40 cores (equivalent of 8 satellites). The limit is reached when the disk I/O bound is hit: all queries run extremely fast, leaving only update transactions in the system to run longer, and facing contention on the disk. The gap between the linear scalability line and Multimed's performance is constant, being caused by the computational resources required by Multimed's middleware.

4.3.2 Increased update workload

Figures 3(b) and 3(c) present the throughput of PostgreSQL (running on different number of cores) and of Multimed (running with different configurations), as the number of clients increases. Note that PostgreSQL has problems in



(a) Scalability throughput: 2GB, 400 clients

(b) Throughput for 2GB database

(c) Throughput for 20GB database

(d) Scalability response time: 2GB, 400 clients

(e) Response time for 2GB database

(f) Response time for 20GB database

Figure 4. PostgreSQL: Standalone vs. Multimed, *Shopping* mix

scaling with the number of clients issuing the workload, and its performance at 48 cores is lower than at 12.

For both dataset sizes, Multimed (at all optimization levels) outperforms the standalone version of PostgreSQL. The $C0$ optimization level for Multimed shows higher error bars, as all satellites are going concurrently to disk, in order to persist updates. Switching to the $C1$ optimization level, we reduce the contention on disk, by using more main memory. We see an improvement of more than 1000 transactions per second between the naïve $C0$ and optimized $C1$ versions of Multimed. Finally, switching to the less generic optimization level $C2$, Multimed accommodates more satellites in the available memory, and can take advantage of the available computational resources, until a disk I/O limit is hit. Using the $C2$ optimization, the problem of load interaction is also solved by routing the "heavy", analytical, queries to different satellite nodes, offloading the other nodes in the system. In all these experiments we have used static routing.

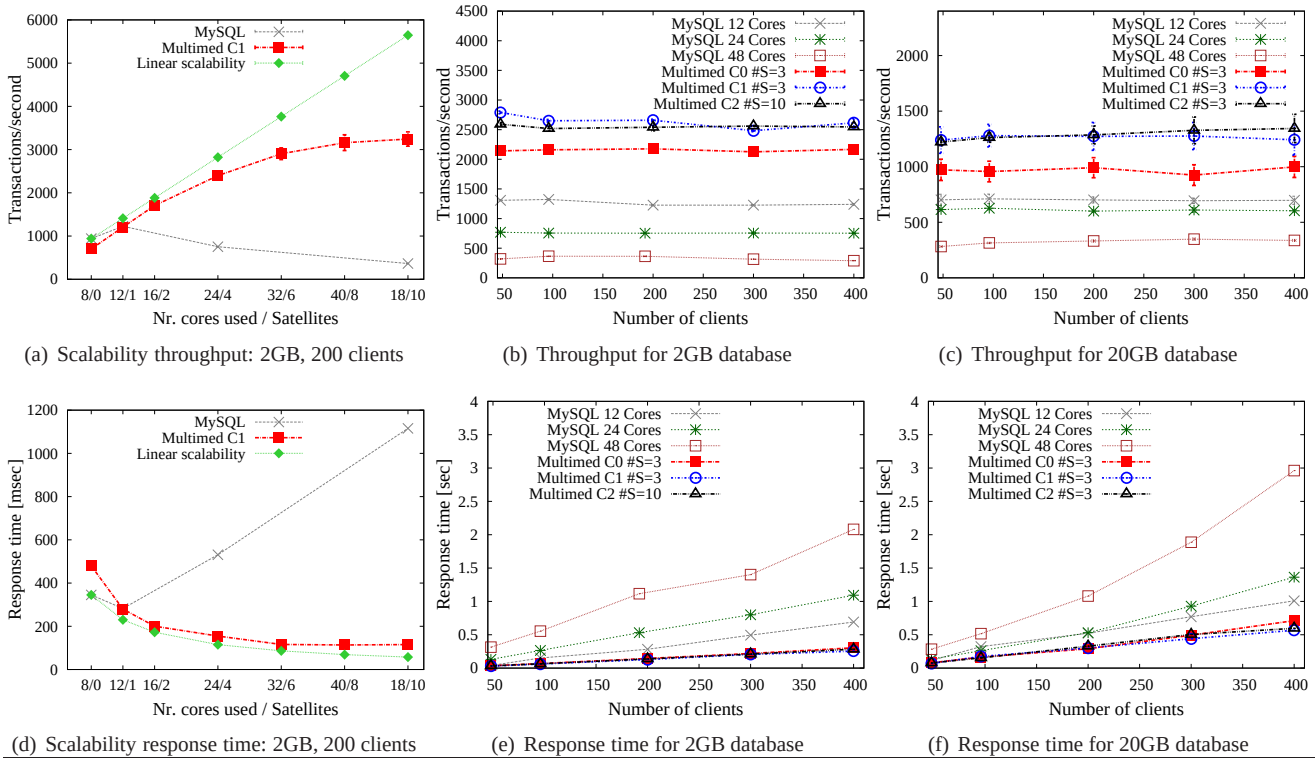
Note the fact that Multimed retains a steady behavior with increasing number of concurrent clients (up to 1000), without exhibiting performance degradation. Looking at the corresponding response times, even under heavy load, Multimed's response time is less than 1 second, indicating that Multimed is not only solving the problems of load interaction, but also the client handling limitations of PostgreSQL. For the Shopping mix, standalone PostgreSQL's performance is slightly better than for the Browsing mix due to the reduced number of heavy queries.

Figures 4(a) and 4(d) show that even in the case of the Shopping mix, PostgreSQL can not scale with the number of available cores, on the 2GB database, with 400 clients. Multimed scales up to 16 cores (2 satellites), at which point the disk becomes a bottleneck. Multimed's performance stays flat with increasing cores, while that of PostgreSQL drops.

Figures 4(b) and 4(c) show that PostgreSQL can not scale with the number of clients for this workload either, regardless of the database size. In the case of Multimed, for a small number of clients, all queries run very fast, leaving the updates to compete for the master node. Past 150 clients, the run time of queries increases and the contention on the master node is removed, allowing Multimed to better use the available satellites. We again observe that Multimed's behavior is steady and predictable with increasing load.

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

Both in the case of a query intensive workload (Browsing mix) and in the case of increased update workload (Shopping



(a) Scalability throughput: 2GB, 200 clients (b) Throughput for 2GB database (c) Throughput for 20GB database

(d) Scalability response time: 2GB, 200 clients (e) Response time for 2GB database (f) Response time for 20GB database

Figure 5. MySQL: Standalone vs. Multimed, *Browsing* mix

mix), PostgreSQL does not scale with the number of cores or with the number of clients, regardless of the database size. PostgreSQL's inability to scale with the number of clients is due to the fact that for each new client a new process is spawned on the server. This might lead to the conclusion that the number of processes is far greater than what the operating system and the hardware can handle. This is disproved by Multimed, which can cope with 1000 clients in spite of the limitations of PostgreSQL. The problem in this case is not the large number of processes in the system, but rather the inability of a single PostgreSQL engine to handle high concurrency. Since Multimed splits the number of clients over a set of smaller sized satellites, it reduces the contention in each engine, resulting in a higher throughput

4.4 MySQL: Standalone vs. Multimed version

In this section we compare standalone MySQL to Multimed running on top of MySQL computational nodes.

For MySQL, we have done our experiments using its InnoDB storage engine. This engine is the most stable and used storage engine available for MySQL. However it has some peculiar characteristics: (i) it acts as a queuing system, allowing just a fixed number of concurrent threads to operate over the data (storage engine threads); (ii) it is slower than the PostgreSQL engine for disk operations. In all the results presented below, the number of cores available for MySQL is equal to the number of storage engine threads. Being a queuing system, MySQL will not show a degradation in throughput with the number of clients, but rather exhibits linear increase in response time. For this reason, the experiments for MySQL only go up to 400 clients.

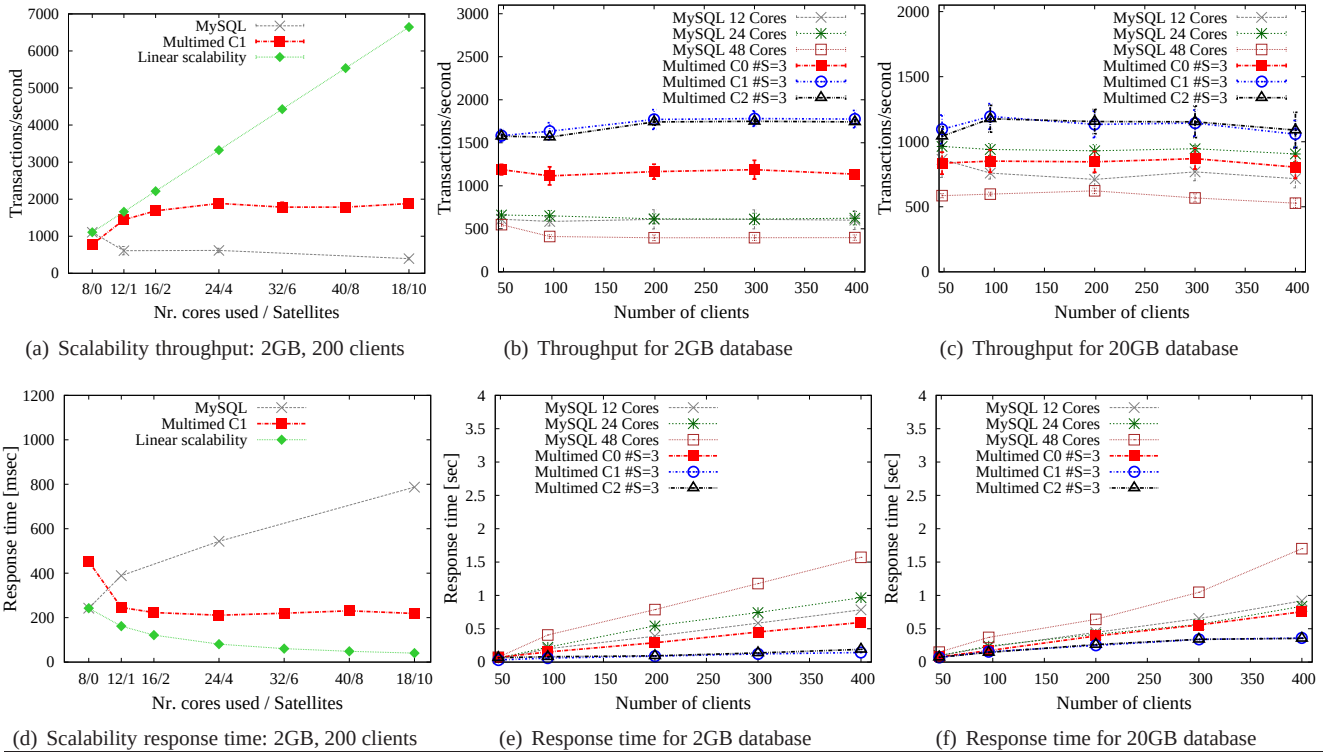
4.4.1 Query intensive workload

Figures 5(a) and 5(d) present the ability of the standalone engine, and of Multimed running on top of it, to scale with the amount of computational resources, in the case of the 2GB database and 200 clients. The x-axis, as before, indicates the total number of cores available for MySQL and Multimed, as well as the number of satellites coordinated by Multimed. Each satellite runs on 4 cores.

In the case of the TPC-W Browsing mix, we notice that MySQL does not scale with the number of cores. Figure 5(a) shows that MySQL performs best at 12 cores. Adding more cores increases contention and performance degrades.

The same conclusion can be seen in the throughput and response time plots for both the 2GB and 20GB datasets (figures 5(b) and 5(c)), that show the performance of MySQL (running on different number of cores) and of Multimed (running on different configurations) with increasing clients. Since the behavior is independent of the dataset, we conclude that the contention is not caused by a small dataset, but rather by the synchronization primitives (i.e., mutexes) that are used by MySQL throughout its entire code.

In contrast, Multimed scales with the number of cores. Figure 5(a) shows that on the 2GB dataset, Multimed scales up to 6 satellites, at which point the disk I/O becomes the



(a) Scalability throughput: 2GB, 200 clients (b) Throughput for 2GB database (c) Throughput for 20GB database

(d) Scalability response time: 2GB, 200 clients (e) Response time for 2GB database (f) Response time for 20GB database

Figure 6. MySQL: Standalone vs. Multimed, *Shopping* mix

bottleneck in the system, and the throughput and response times are flat. The fact that Multimed on top of PostgreSQL scaled in the same test up to 8 satellites corroborates the fact the PostgreSQL's storage engine is faster than MySQL's InnoDB for this workload.

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

4.4.2 Increased update workload

The scalability plot (figure 6(a)), shows that MySQL performs best at 8 cores. With more cores performance degrades, confirming that contention is the bottleneck, not disk I/O. Multimed scales up to 16 cores, at which point the throughput flattens confirming that the disk becomes the bottleneck.

Figures 6(b) and 6(c) show that on larger datasets data contention decreases, allowing standalone MySQL to perform better. On the 2GB database, Multimed brings an improvement of 3x. In the case of the 20GB database, Multimed achieves a 1.5x improvement.

5. Discussion

Multimed adapts techniques that are widely used in database clusters. As a database replication solution, Multimed inherits many of the characteristics of replicated systems and database engines. In this section we discuss such aspects to further clarify the effectiveness and scope of Multimed.

5.1 Overhead

The main overhead introduced by Multimed over a stand alone database is latency. Transactions are intercepted by Multimed before being forwarded either to the master or to a satellite. In the case the transactions go to a satellite, there might be further delay while waiting for a satellite with the correct snapshot.

Multimed works because, for a wide range of database loads, such an increase in latency is easily compensated by the reduction in contention between queries and the increase in the resources available for executing each query. Although satellite databases in Multimed have fewer resources, they also have less to do. For the appropriate workloads, Multimed is faster because it separates loads across databases so that each can answer fewer queries faster than a large database can answer all the queries.

5.2 Loads supported

There is no database engine that is optimal for all loads [Stonebraker 2008]. Multimed is a replication based solution and, hence, it has a limitation in terms of how many updates can be performed as all the updates need to be done at the master. Although this may appear a severe limitation, it is not so in the context of database applications.

As the experiments above show, Multimed provides substantial performance improvements for the TPC-W browsing and shopping mixes. For the ordering mix, with a higher rate of updates, Multimed offers similar performance as the single database since the bottleneck in both cases is the disk. Multimed can be used to linearly scale read dominated loads such as those found in business intelligence applications and data warehousing. For instance, it is possible to show linear scale up of Multimed by simply assigning more satellites to complex analytical queries. As a general rule, the more queries and the more complex the queries, the better for Multimed.

Workloads with high update rates and without complex read operations are less suitable for Multimed – and indeed any primary copy replication approach – because the master becomes the bottleneck (regardless of why it becomes the bottleneck: CPU, memory, or disk). In cluster based replication, this problem is typically solved by simply assigning a larger machine to the master. Multimed can likewise be configured to mitigate this bottleneck with a larger allocation of resources (cores, memory) to the master.

5.3 Configuration

Tuning and configuring databases is a notoriously difficult problem. Some commercial database engines are known to provide thousands of tuning knobs. In fact, a big part of the impetus behind the autonomic computing initiative of a few years ago was driven by the need to automatically tune databases.

Similarly, tuning Multimed requires knowledge of database loads, knowledge of the engines used, and quite a bit of experimentation to find the right settings for each deployment. The advantage of Multimed over a stand alone database is that the number of global tuning knobs is less as each element of the system needs to be tailored to a specific load. The master can be tuned for writes, the satellites for reads. It is even possible to configure Multimed so that a satellite answers only specific queries (for instance, particularly expensive or long running ones) and then optimize the data placement, indexes, and configuration of that satellite for that particular type of query.

In terms of the interaction with the operating system and the underlying architecture, Multimed can be configured using simple rules: allocate contiguous cores to the same satellites, restrict the memory available to each satellite to that next to the corresponding cores, prevent satellites from interfering with each other when accessing system resources (e.g., cascading updates instead of updating all satellites at the same time), etc. Such rules are architecture specific but rather intuitive. Note as well that Multimed is intended to run in a database server. These are typically powerful machines with plenty of memory, often fast networks and even several network cards, and SAN/NAS storage rather than local disks. The more main memory is available, the faster the I/O, and the more cores, the more possibilities to tune Multimed to the application at hand and the bigger performance gains to be obtained from Multimed.

5.4 Optimizations

The version of Multimed presented in the paper does not include any sophisticated optimizations since the goal was to show that the basic concept works. There are, however, many possible optimizations over the basic system.

WriteSet extraction is currently done through triggers. Depending on the system and load, this can become a bottleneck (although note that the master where the trigger runs no longer handles any query, so it has extra capacity to run the triggers). An alternative is to extract the changes directly from the log buffers (which requires access to the database internals but has been done before in a number of systems). Another option is to capture the changes from the log, a common solution in many database replication products such as Data Streams of Oracle.

As mentioned, using a SAN/NAS storage will significantly speed up Multimed because the I/O bottleneck can be greatly reduced. Similarly, as has been done in some commercial systems like Oracle Exadata, one can use SSDs as caches between the remote disks and the server to minimize the I/O impact. In the same way that these solutions speed up traditional databases, Multimed will be faster using them.

The extra communication cost incurred by Multimed is very small compared to the total run time of each transaction. Even so, we have presented in section 3.4 a couple of tweaks that we have performed in order to reduce it even more.

Like any replicated database, Multimed requires additional resources for every copy of the database. In a multicore machine, the number of satellites that can be used depends on the available memory. The situation can be improved by using partial replication as we have shown in this paper. If each satellite contains only part of the original database, then the available memory allows for the creation of more satellites (subject to the deployment constraints discussed above). Multimed offers many options for exploring such settings: satellites can be tailored for answering certain queries (which only need the data necessary to answer those queries), or satellites with special indexes for given workloads. Satellites could also be enhanced with user defined functions, offering a way to extend the functionality of system without having to modify the master database. Several such optimizations have been shown to work well in cluster based system and can easily be adapted to run in Multimed [Plattner 2008].

6. Related work

Multimed builds on work done in database replication and multikernel operating systems.

6.1 Database Replication

Multimed uses many ideas from database replication strategies developed during the last decade starting with the work on Postgres-R [Kemme 2000]. Of the many existing systems [Cecchet 2008], however, not all approaches are suitable for multicore machines. For instance, many middleware database replication solutions use group communication to coordinate the copies [Bettina 2003; Elnikety 2006]. These solutions require a multi-master configuration and rely on full replication. As our experiments have shown, running the satellites in main memory and being able to implement partial replication is a great boost in performance. Nevertheless, some of the innovative applications pursued with these systems could also be implemented using the Multimed approach within a single multicore machine rather than on a cluster. An example is the work on tolerating byzantine failures using heterogeneous replicas [Vandiver 2007] or commercial version used in data warehousing [Xkoto].

The approaches closer to the design of Multimed are those relying on single master replication [Daudjee 2006; Plattner 2004] and that can support specialized satellites and partial replication. This type of design is starting to be widely used in cloud computing, for instance, in the Microsoft SQL Azure database [Campbell 2010].

6.2 Multikernel Operating Systems

The problems that multicore creates in system software either because of the increasing number of cores [Agarwal 2007; Borkar 2007] or their potential heterogeneity [Hill 2008; Kumar 2004] are by now well known. This has triggered a lot of activity in the area of operating systems to address these problems. For instance, [Wickizer 2008] proposes a new exokernel based operating system, Corey, which tries to manage the complexity of multicore machines by moving the responsibility into the application space. Disco [Bugnion 1997] and Cellular Disco [Govil 2000], make the case for resource partitioning by running a virtualization layer over a shared memory architecture, allowing the execution of multiple commodity operating systems, and treating multicore as a cluster. Finally, [Baumann 2009; Nightingale 2009; Wentzlaff 2009] make a clean statement that multicore machines should be viewed as distributed systems and adequate algorithms and communication models should be employed.

The work in Multimed borrows many of these concepts and applies them to the special (and architecturally very different) case of database engines.

7. Conclusions

In this paper we address the problem of making databases run efficiently on multicore machines. Multimed, the system we present in the paper, represents a departure from existing work in that it solves the problem for a wide range of loads without having to modify the engine. Instead, it uses existing databases in a replicated configuration and deploys them over a multicore machine as if the multicore machine were a distributed system. As shown in the evaluation, Multimed exhibits better and more stable performance on multicore architectures than PostgreSQL and MySQL.

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

8. Acknowledgments

We would like to thank the reviewers for their helpful comments and insights.

In particular, we would like to thank Rebecca Isaacs from Microsoft Research for her help in preparing the final version of the paper.

References

[Agarwal 2007] A. Agarwal and M. Levy. The kill rule for multicore. In *IEEE DAC '07*, pages 750–753, San Diego, California, 2007.

[Alonso 2011] G. Alonso, D. Kossmann, and T. Roscoe. SwissBox: An Architecture for Data Processing Appliances. In *CIDR '11*, pages 32–37, Asilomar, California, 2011.

[Apache Mina] Apache Mina. `http://mina.apache.org/`.

[Baumann 2009] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM SOSP '09*, pages 29–44, Big Sky, Montana, 2009.

[Bettina 2003] K. Bettina. *Future directions in distributed computing*. Springer-Verlag, 2003.

[Borkar 2007] S. Borkar. Thousand core chips: a technology perspective. In *IEEE DAC '07*, pages 746–749, San Diego, California, 2007.

[Boyd-Wickizer 2010] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M.F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *USENIX OSDI '10)*, pages 1–8, Vancouver, Canada, 2010.

[Bryant 2003] R. Bryant and J. Hawkes. Linux Scalability for Large NUMA Systems. In *Linux Symposium*, pages 83–96, Ottawa, Canada, 2003.

[Bugnion 1997] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM TOCS '97*, 15(4):412–447, 1997.

[Campbell 2010] D.G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *ACM SIGMOD Conference*, pages 1021–1024, 2010.

[Cecchet 2008] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *ACM SIGMOD 08*, pages 739–752, Vancouver, Canada, 2008.

[Daudjee 2006] K. Daudjee and K. Salem. Lazy Database Replication with Snapshot Isolation. In *VLDB '06*, pages 715–726, Seoul, Korea, 2006.

[Elnikety 2006] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *ACM EuroSys '06*, pages 117–130, Leuven, Belgium, 2006.

[Govil 2000] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM TOCS 2000*, 18(3):229–262, 2000.

[Gray 1996] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD '96*, pages 173–182, Montreal, Canada, 1996.

[Han 2009] W.S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core CPUs. In *ACM SIGMOD '09*, pages 45–58, Providence, Rhode Island, 2009.

[Hardavellas 2007] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR '07*, pages 79–87, Asilomar, California, 2007.

[Harizopoulos 2005] S. Harizopoulos and A. Ailamaki. StagedDB: Designing database servers for modern hardware. In *IEEE ICDE '05*, pages 11–16, Tokyo, Japan, 2005.

[Hill 2008] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. In *IEEE HPCA '08*, volume 41, pages 33–38, Salt Lake City, Utah, 2008.

[Johnson 2009a] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In *DaMoN '09*, pages 21–26, Providence, Rhode Island, 2009.

[Johnson 2009b] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *ACM EDBT '09*, pages 24–35, Saint Petersburg, Russia, 2009.

[Kemme 2000] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB 2000*, pages 134–143, Cairo, Egypt, 2000.

[Kumar 2004] R. Kumar, D.M. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *IEEE ISCA '04*, volume 32, pages 64–75, Munich, Germany, 2004.

[Liu 2009] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: space-time partitioning in a manycore client OS. In *USENIX HotPar'09*, pages 10–10, Berkley, California, 2009.

[Nightingale 2009] E.B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *ACM SOSP '09*, pages 221–234, Big Sky, Montana, 2009.

[OProfile] OProfile. `http://oprofile.sourceforge.net/`.

[Papadopoulos 2008] K. Papadopoulos, K. Stavrou, and P. Trancoso. HelperCore DB: Exploiting multicore technology to improve database performance. In *IEEE IPDPS '08*, pages 1–11, Miami, Florida, 2008.

[Peterson 2000] L. Peterson and B.S. Davie. *Computer networks: a systems approach*. Morgan Kaufmann Publishers Inc., 2000.

[Plattner 2004] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *ACM Middleware*, pages 155–174, Toronto, Canada, 2004.

[Plattner 2008] C. Plattner, G. Alonso, and M. Tamer Özsu. Extending DBMSs with satellite databases. *The VLDB Journal*, 17 (4):657–682, 2008.

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

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

[Vandiver 2007] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *ACM SOSP '07*, pages 59–72, Stevenson, Washington, 2007.

[Veal 2007] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *ACM ANCS '07*, pages 57–66, Orlando, Florida, 2007.

[Wentzlaff 2009] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM Operating Systems Review*, pages 76–85, 2009.

[Wickizer 2008] Silas B. Wickizer, H. Chen, Y Mao, M. Frans Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, et al. Corey: An Operating System for Many Cores. In *USENIX OSDI '08*, San Diego, California, 2008.

[Xkoto] Xkoto. `http://www.xkoto.com/`.