RapiLog: Reducing System Complexity Through Verification

Gernot Heiser Etienne Le Sueur
Adrian Danis Aleksander Budzynowski
NICTA and UNSW, Sydney, Australia
gernot@nicta.com.au

Tudor-Ioan Salomie Gustavo Alonso
ETH Zurich, Switzerland
{tsalomie,alonso}@inf.ethz.ch

Abstract

Database management systems provide updates with guaranteed durability in the presence of OS crashes or power failures. Durability is achieved by performing synchronous writes to a transaction log on stable, non-volatile storage. The procedure is expensive and several techniques have been devised to ameliorate the impact on overall performance at the cost of increased system complexity.

In this paper we explore the possibility of reducing the system complexity around logging by leveraging verification instead of using specialised/dedicated hardware or complicated optimisations. The prototype system, *RapiLog* , uses a dependable hypervisor based on seL4 to buffer log data outside the database system and its OS, and performs the physical disk writes asynchronously with respect to the operation of the database. RapiLog guarantees that the log data will eventually be written to the disk even if the database system or the underlying OS crash or electrical power is cut. We evaluate RapiLog with multiple open-source and commercial database engines and find that performance is never degraded (beyond the virtualisation overhead), and at times is significantly improved.

1. Introduction

Databases in general and relational database engines in particular are a cornerstone of enterprise applications. They provide an efficient way to store, manage and query large amounts of data, freeing developers from having to implement data management functionality at the application level.

Critical to the operation of relational database engines is the concept of a *transaction*—the reliable unit of work (reading or writing information) done in a database that either entirely completes or has no effect at all. When performing transactions, the *database management system* (DBMS) is required to enforce the **ACID** properties of *atomicity*, *consistency*, *isolation* and *durability* (Härder and Reuter 1983).

In here, we are mainly interested in **durability**, a property achieved by forcing all changes that occur in the database to be first stored on a stable non-volatile medium before they become visible outside the corresponding transaction. Commonly, a *transaction log* is used to ensure that committed transactions can survive system failures. In the event of a crash, once the system is recovered, the database state can be reconstructed from the log.

The failure model used by DBMSes typically assumes a perfect persistent storage (often achieved using RAID techniques) but allows for power failures and fail-stop crashes of the operating system (OS) to happen at any time. Commonly, the durability of the transaction log is provided through write-ahead logging (Mohan et al. 1992): when a client attempts to commit a transaction, the DBMS stores the intended changes to the database state in an append-only log and the transaction is only assumed to be committed once it is known that the log data is recorded on persistent storage. Persistence is achieved by using synchronous (e.g. `fsync`) operations on the log disk for transactions or groups of them.

However, synchronous writes to the transaction log limit the performance of a DBMS to the speed of the log disk. To ameliorate the situation, several techniques are used in practice involving complex system designs (group commit, log multiplexing) and/or specialised hardware (SSD logs). Often, latency can be reduced and throughput significantly increased if the log is written asynchronously, at the expense of durability (see Figure 2, Section 2.3). For this reason, commercial engines offer control over logging policy, to let the user decide on the trade-off between durability and performance.

In this paper we explore the option of using verified software with strong guarantees instead of complex optimisation or specialised hardware. We propose RapiLog, a system supporting asynchronous log writes without sacrificing durability. RapiLog buffers log writes so they can be written to stable storage asynchronously to transaction processing. The buffering is transparent to the DBMS and the OS hosting it.

RapiLog leverages recent advances in dependable

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04. . . $15.00

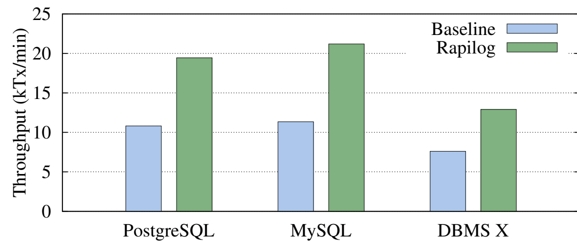

Figure 1. Performance of RapiLog vs standard logging optimisation techniques as implemented in 3 different database engines. "Baseline" refers to the "virtualised" configuration in Figure 9; higher is better.

systems, specifically the formal verification of OS kernels (Klein et al. 2009) and synthesis of device drivers (Ryzhyk et al. 2009) to protect the database log data from system failures. Combined with a technique for dealing with power failures, RapiLog provides an alternative design to transaction logging. It reduces the overall system complexity as it offloads most of the intricacies of durable transaction operations outside of the DBMS.

RapiLog not only operates without performance loss (beyond virtualisation overhead): we actually observe significant performance improvements for 3 different DBMS engines (one of them a highly optimised commercial database), as shown in Figure 1 (an excerpt from Figure 9 where the complete results are presented).

This paper makes the following contributions:

- The demonstration that a robust system, relying on verification and synthesis of its components, can replace costly hardware and complex system optimisations.
- The design and implementation of RapiLog, based on these principles: by design, RapiLog enables asynchronous writes to a DBMS's transaction-log without compromising durability (Section 3), while its implementation transparently provides this benefit to unmodified legacy DBMSes (Sections 4.1–4.2).
- A thorough evaluation of RapiLog on three DBMSes, showing its performance benefits under typical workloads (Sections 5.3–5.5), and its ability to maintain transaction durability on external power failure by leveraging the energy stored in power-supply capacitances (Section 5.7).

2. Logging as a Performance Bottleneck

2.1 Durability in database engines

Consistency in a database is defined in terms of transactions: a correct transaction executed in its entirety takes the database from one consistent state to another consistent state. Conventional database engines implement consistent transactions by enforcing the *ACID* properties: atomicity, consistency, isolation, and durability.

Atomicity ensures that transactions are executed in their entirety and translates into mechanisms that will remove any changes introduced by a transaction if the transaction does not commit.

Consistency ensures that a transaction performs only legal operations and it is enforced mainly through integrity constraints over the schema.

Isolation ensures that transactions work only on a consistent state of the database. Since the intermediate database states created as transactions execute are not guaranteed to be consistent, concurrency control mechanisms are used to isolate a transaction from other concurrent transactions.

Durability enforces the long-term memory of the database: every transaction that commits is persistently stored. In the event of a system failure, durability ensures that the state of the database and the effects of every committed transaction can be restored.[1]

In here, we will focus on durability.

2.2 Durability and logging

Durability relies on a variety of logging and recovery schemes (e.g., physical vs. logical logging, and undo vs. redo log records (Bernstein et al. 1987)) with commercial products providing a range of guarantees depending on what is logged (e.g., in Microsoft SQL-Server (Microsoft 2012) the *full mode* guarantees full recovery, while other modes provide less durability).

Most relevant for durability are redo logs (Lomet 1998; Lomet and Tuttle 2003). By default, the redo log records of a transaction must be written to disk before the database can confirm that a transaction has been committed. The records need to be written in the order the transactions commit and, as a result, the latency of the transaction is often determined by the latency of writing to disk.

Logging adds to the complexity of critical code. Furthermore, if durability is to be enforced, the throughput of the database is limited by the I/O bandwidth to the log, which can become a bottleneck for OLTP-style workloads (DeWitt et al. 1984). Not surprisingly, trying to speed up database transaction logs has been a very active area of research, with many optimisations having been proposed in the literature over the years (Gray and Reuter 1993). We will just mention two of the most widely used ones, which illustrate well the added complexity incurred in speeding up logging.

A first optimisation is *group commit*. Instead of forcing to disk the records of each transaction individually, commit requests can be slightly delayed and the records of several transactions flushed in one single operation. Synchronous logging issues an `fsync` system call after each group of transactions. Group commit improves the efficiency of the log disk by amortising the cost of setting up a write and the

[1] Databases typically provide an even stronger property: point in time recovery, meaning that they can recover to any point in the committed history of the database.

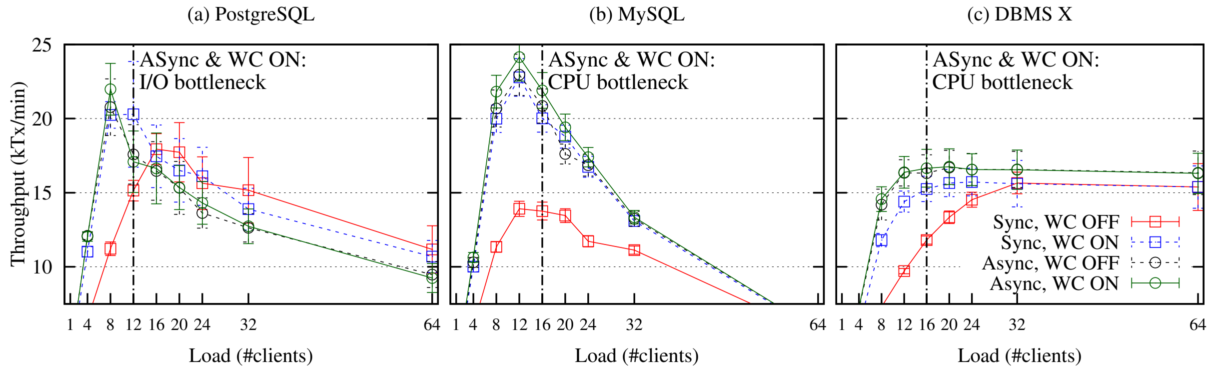

Figure 2. Throughput of three databases under the same offered load, with synchronous and asynchronous log writes and disk cache on or off. Only the *Sync, WC OFF* configuration ensures durability. Error bars indicate standard deviations.

subsequent seek operations.

A second optimisation is *log multiplexing*, where the actual log is split among multiple (typically two) disks. Commit entries for different transaction groups are written to different logs, so a group does not have to wait for the previous one to complete.

2.3 The compromise: Asynchronous commits

Because durability is expensive, most engines today offer the option to turn logging off to reduce the amount of data logged or to make logging asynchronous. In all cases, performance improves at the cost of potential losses in case of failures. With asynchronous logging or *asynchronous commits*, transactions are allowed to commit before their log records are recorded on disk – risking to lose the most recent transactions in the case of a crash. Hardware support, in the form of non-volatile memory, battery-backed caches or uninterruptible power supplies (UPS) can be used to minimise the potential losses.

Due to the complications that arise in case of failures, databases restrict when a commit can be made asynchronous. Oracle, for instance, does not allow asynchronous commits for distributed transactions (Oracle 2000), as it leads to inconsistencies across nodes.

Another way of minimising the impact on performance of synchronous logging is to turn on the disk's write cache: write requests get buffered on the disk drive rather than fully flushed to the platters. Being another form of asynchrony, this also sacrifices durability (McKusick 2006; Nightingale et al. 2008) unless one ensures that these writes will end up on disk and not be lost in the volatile disk cache in the case of power failures.

The performance gain of using asynchronous writes can be substantial. Figure 2 presents the throughput of three different databases when using synchronous and asynchronous writes and when switching the disk cache on and off. The databases are PostgreSQL, MySQL and a commercial engine, dubbed DBMS X.[2] All three systems are hosted on Linux, and we use separate disks for the transaction log and the actual database data. This is a widely-used configuration, and allows us to evaluate the impact of logging independent of other I/O issues (see Section 5 for details). The three engines show the same behaviour: enabling either asynchronous commits (*Async*) or the disk's write cache (*WC ON*), leads to reduced commit latency and, in turn, to higher throughput. The only configuration that offers durability (*Sync, WC OFF*), exhibits on all the engines significantly lower throughput than the other configurations under the same load conditions.

Figure 2 provides interesting insights into the behaviour of the different engines. The dashed vertical lines in Figure 2 show the point at which a bottleneck is reached, and the database goes into overload. Database operators try to configure their system such that overload is avoided, as it results in significantly increased transaction latencies.

PostgreSQL hits an I/O write bottleneck on the data disk at 12 clients. MySQL and DBMS X hit a CPU bottleneck (more than 90% CPU utilisation) at 16 clients. DBMS X, being a highly optimised commercial engine, is the only database that offers stable performance even after the bottleneck is reached, with performance degrading significantly for both PostgreSQL and MySQL in overload. Note also that PostgreSQL performance fluctuates significantly under overload conditions, as indicated by the large error bars in Figure 2 (a) beyond 12 clients. This is a result of the data disk bottleneck: as data writes are essentially random, there are random opportunities for the disk to coalesce or merge sequential write requests. As the bottleneck of MySQL and DBMS X is caused by CPU load, performance remains predictable (evidenced by the small error bars).

2.4 Asynchronous commits without losses?

As shown in Figure 2, a significant increase in performance can be gained by disabling `fsync` (*Async*) or enabling the

[2] The license of the commercial system prohibits us from identifying it.

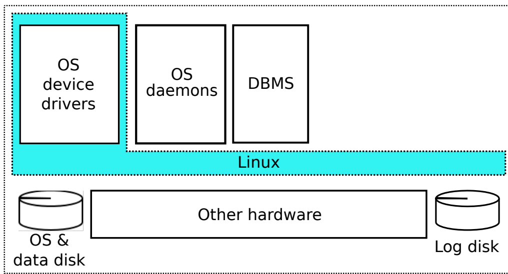

Figure 3. Standard DBMS system architecture.

transaction log disk's write cache (*WC ON*). However, the performance improvements come at the expense of durability: data can be permanently lost and transactions or data integrity can no longer be guaranteed if the OS crashes or power fails at an inopportune moment, leaving the database in an inconsistent state. Recovering from such failures (if no data was lost) without a consistent transaction log would require manual intervention. Such manual recovery of the database is a complex, expensive, and cumbersome task that leaves the system unavailable for the duration of the recovery process.

RapiLog aims to achieve the best of both worlds: reducing complexity while maximising throughput by using asynchronous logging, but at the same time retaining durability by guaranteeing that all log entries of committed transactions will *eventually* be recorded on persistent storage, even when OS or power failures occur. RapiLog makes log writes *logically synchronous but physically asynchronous*.

The RapiLog prototype offers almost the same database durability guarantees as a traditional setup for the transaction log: it tolerates DBMS and OS crashes as well as power failures, but (like a normal DBMS) does not protect against OS bugs corrupting data before it reaches the disk. We are presently weakening the guarantees in one point: traditional DBMSes are protected against fail-stop hardware failures, such as machine checks, which RapiLog does not yet handle (but see the discussion in Section 7 on how this could be approached).

3. RapiLog

Figure 3 shows how a typical database server is deployed on a machine. Obviously, the simple "disks" may be large storage arrays or network-attached storage.

In the standard architecture, the DBMS relies on the OS (for example, Linux) to provide access to disks through device drivers within the OS kernel. Often that is the only involvement of the OS in the I/O traffic between database and disk, as the database bypasses the OS and the file system. Being a monolithic kernel, all device drivers on the system run at the same privilege level and in the same address space; hence if a driver crashes, it can bring the whole system down.

The core idea behind RapiLog is to reduce the complexity and latency cost of durability by allowing the system to write

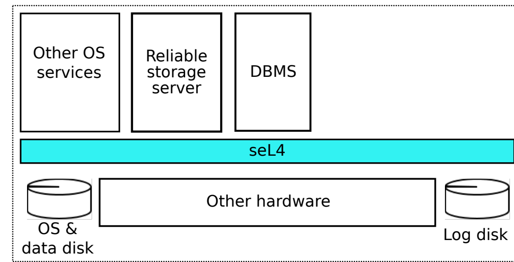

Figure 4. Architecture of an ideal RapiLog implementation, with the DBMS running natively on top of the dependable microkernel, and an isolated component providing a reliable storage service.

transactions to the log asynchronously. RapiLog achieves this by avoiding the `fsync` operation which forces data to be written to the log disk before allowing the DBMS to continue operation. By eliminating synchronous I/O, the system is less limited by the speed at which the log disk can complete writes. Durability is not impacted, as long as we can guarantee that the log data will eventually be recorded on the disk, even in the case of a complete OS, DBMS or power failure. In RapiLog we buffer the log data *outside* the main OS, and achieve durability as long as we can guarantee that the software, which maintains and protects the buffer and effects its recording on disk, will operate correctly in the event of failures of other system components.

3.1 RapiLog architecture

Figure 4 depicts an ideal scenario, with a DBMS running as a native microkernel application, supported by microkernel-based OS services. The log buffer and the software managing it are encapsulated in a separate address space and use a separately encapsulated reliable storage server to write the log data to disk. The formally-verified microkernel is guaranteed to never crash, even if user-level code misbehaves in an arbitrary way. We can assure the correct behaviour of the buffering software by traditional software engineering means (testing and code inspection) or by formal verification (linked to the formal specification of the microkernel (NICTA 2011)).

This architecture, besides supporting asynchronous logging, can lead to a simplified DBMS implementation (potentially resulting in further performance gains) by leveraging the reliability of the underlying kernel. The proposed approach offers (a) the opportunity of eliminating a significant part of the (complex) code devoted to these optimisations; and (b) reducing the burden on database administrators to decide how to configure and use the transaction log. In principle, we could extend the approach to the page/block management of the database, aiming to reduce its complexity as well as removing I/O overheads caused by high update rates. It also supports the use of I/O services optimised for DBMS needs (Stonebraker 1981).

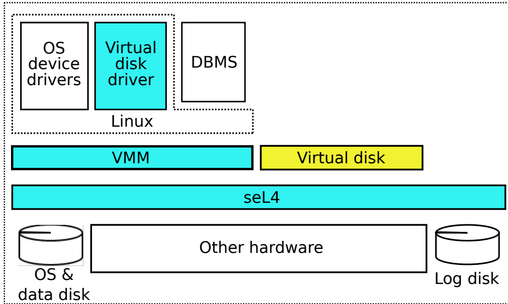

Figure 5. Architecture of the prototype RapiLog implementation using an unmodified DBMS and a virtualised guest operating system, Linux.

The drawback of the architecture in Figure 4 is the engineering effort required for porting an existing DBMS. It is therefore most suited when developing a DBMS from scratch. Given the huge investment in legacy DBMSes, an approach which aids retrofitting an existing DBMS for asynchronous logging would be preferable.

Figure 5 shows the design of the prototype RapiLog system, which supports unmodified legacy DBMSes (Heiser et al. 2011). At the lowest level, we use the seL4 microkernel as a hypervisor on which to run a guest OS in a virtual machine. A *virtual-machine monitor* (VMM) handles interrupts and guest requests. Alongside the guest OS runs a *virtual disk* (VD), which provides access to the transaction log disk. A custom block driver (the *virtual disk driver*, VD driver), loaded into the guest OS, relays I/O requests to the VD through microkernel IPC. Synchronisation requests issued by the guest force the VD driver to drain the guest's I/O queue to the VD but not necessarily trigger physical I/O.

By using this approach, any unmodified, legacy DBMS, running on top of an unmodified guest OS (other than a custom block driver) can obtain the benefits of RapiLog.

Our approach bears some semblance with Rio Vista (Lowell and Chen 1997), which buffers transaction logs in memory and uses the Rio file cache (Chen et al. 1996) to protect the buffered log from OS crashes. Rio is part of the OS, and as such not fully protected from corruption, but achieves good isolation though mapping OS code read-only and minimising dependence on OS data structures through the use of physical addressing and leveraging BIOS disk drivers.

3.2 Virtual disk architecture

The VD, as shown in more detail in Figure 6, provides the RapiLog functionality. It essentially consists of the log buffer (*Memory buffers*), a real disk driver (*Disk driver*), and some minimal amount of management software (*Virtual disk process*). Log data sent to the VD by the DBMS is stored in the buffer, and once recorded there, the VD driver informs the OS that data is recorded on persistent storage. As a consequence, any `fsync` on the VD should only take as long as a call to the hypervisor and a `memcpy`.

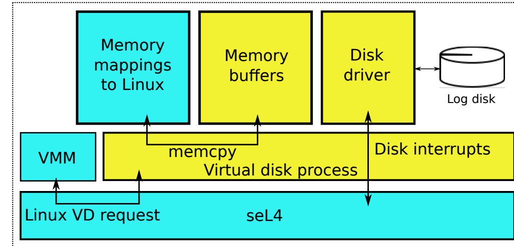

Figure 6. Virtual disk architecture.

I/O completion will only be delayed (and an `fsync` would only block) if the log buffer is full, in which case completion notification will be deferred until some buffer data has been flushed to disk.

3.3 Disk driver

It is common practice in DBMSes to keep the transaction log on a device separate from the actual database. In the architecture of Figure 5, this means that the database is kept on a physical device which is under full control of the guest OS (via a pass-through driver hosted inside the OS). The log is kept on the second disk, which is under the control of the VD. This approach is advantageous to RapiLog, as it means that the log disk is used (almost) exclusively for sequential writes. It implies that the log disk driver need only support minimal functionality and can be optimised for sequential writes, making for a very simple (and thus reliable) driver.

We must note, however, that RapiLog imposes stronger dependability requirements on its disk driver than a standard setup. The scenario of Figure 3 can tolerate fail-stop crashes of the driver (as any OS component) at any time. RapiLog, in turn, must guarantee that the log data will be flushed to disk (eventually) even in the presence of crashes. This means that the disk driver must not fail while there is still dirty log data in the buffer. Hence it is important to have a highly reliable disk driver.

Driver reliability is greatly enhanced by the potential simplicity mentioned above. Even more attractive is the Termite approach (Ryzhyk et al. 2009), which synthesises device drivers from formal specifications of their interfaces, practically eliminating the chance of introducing bugs when coding the driver logic. A sufficiently simple driver even lends itself to formal verification (Alkassar and Hillebrand 2008). Therefore we advocate the use of verified or synthesised drivers for RapiLog.

3.4 Operation during power failure

The logging approach traditionally used in DBMSes has the advantage that it also guards against corruption or data loss resulting from power failures (provided that the disk does not corrupt data). RapiLog, as discussed so far, would lose data if the mains power is cut.

Protection against interruptions to the electrical power

supply does not tend to be a major issue for large database installations, as data centres tend to deploy UPSes. These provide some notification of imminent power loss, giving time for machines to be gracefully shut down, hence avoiding data loss. However, installing UPSes requires a significant investment and, where the database is not expected to remain operational through a power outage, this extra cost should be avoided. Furthermore, a UPS needs to store significant amounts of energy in batteries which are not 100% efficient, losing some charge over time. Hence there is an on-going energy overhead of keeping the batteries charged.

Disk write caches add another dimension to the problem of power loss. Most contemporary disk drives have a cache which can improve write performance by caching writes on the disk itself. Additionally, the disk drive may reorder some writes to improve write performance. Disk drives that report write completions before the data actually makes it to the disk platter can create problems for consistency: In the event of a power failure, the write cache may not be completely written to the disk and data could be lost (McKusick 2006). Thus, DBMS tuning guides recommend disabling disk drive write caches to ensure durability.

The problem can be mitigated another way, through the use of a *battery-backed write cache* (Ousterhout and Douglis 1989), where the controller to which a disk drive is attached provides the write cache instead of the disk drive itself, and also provides some battery-backed, non-volatile storage to act as the write cache. When power is restored, the data in the battery-backed write cache can be written to the disk and operation can continue normally. However, battery-backed caches not only add to the cost of the disk, they also require maintenance due to the limited life time of the batteries.

With the help of some very simple hardware support, RapiLog can protect the DBMS from the effects of power failures making a UPS redundant as far as maintaining the *ACID* properties are concerned. The only requirement is a notification from the hardware to the VD (via the hypervisor) when a power outage is detected. Such an outage does not result in the system shutting down immediately—capacitors in the internal power supplies of computers store enough energy to keep the system operational for a short period (typically 100s of milliseconds, see Section 5.6).

This time is sufficient to write a fair amount of data to disk (at least several MiB). By limiting the size of the buffer, we can ensure that all log data can be safely written to disk when a power outage is detected. In this case, RapiLog invokes hypervisor mechanisms to prevent any other activities from being scheduled, guaranteeing the VD exclusive access to the machine until it finally shuts down from the loss of power.

Some contemporary server platforms provide notifications of power outages, which RapiLog can use to determine when normal system operation should cease and buffer write back should be prioritised. Additionally, some power distribution units (PDUs) will send SNMP traps when events such as outages or overloads occur, which can be used in similar ways. For platforms that do not provide, or have access to, a power outage notification, a simple device can be used to generate an interrupt when power is lost (we built a prototype from a few standard components in about 20 minutes, shown in Figure 7).

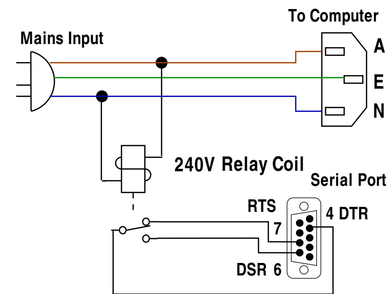

Figure 7. Simple device which can be used to signal power outages.

3.5 Performance benefits

While RapiLog's primary aim is to show the potential for reducing database system complexity, the architecture described here can also improve DBMS performance in a number of ways.

Firstly, RapiLog is designed to avoid blocking I/O. Unlike during normal DBMS operation, where at commit time, transaction processing will block until the commit is logged on disk, RapiLog is designed to let transaction processing proceed concurrently while writing log data to disk. This overlapping of processing with disk writes is the potentially most significant performance benefit of RapiLog. Figure 2, which compares normal DBMS operation (with `fsync`) with operation where `fsync` calls are omitted, gives an idea of the performance gains that can be achieved when asynchronous writes are enabled.

Secondly, RapiLog makes better use of disk bandwidth. Transaction log entries are frequently small. Indeed, many log writes are only a few dozen bytes in size, though this is highly dependent on the DBMS and workload. For example, MySQL's InnoDB engine writes only modified records to the redo-transaction log, while more complex systems like Oracle also pack transaction rollback information in the redo-log entries—considerably increasing their size (Oracle 2008).

As disk writes must transfer whole blocks, a significant portion of the available I/O bandwidth can potentially be wasted when the disk is forced to write partial blocks. RapiLog, in contrast, can defer disk writes until at least one full block of log data is available for writing, in the case when redo-log entries are smaller than the block size. During high load, this batching behaviour causes the average disk transfer to be larger and so numerous overheads are reduced—fewer seeks are required, less time is spent initiating disk

transactions, and fewer interrupts are generated, relative to the amount of data written.

4. Implementation

4.1 Overview

We implemented the RapiLog architecture using seL4 as the dependable hypervisor. Being a general-purpose microkernel, seL4 supports virtual machines through a user-level virtual-machine monitor (VMM), similar to the approach taken by NOVA (Steinberg and Kauer 2010).

For performance reasons, as well as to keep the implementation as simple as possible, we make extensive use of memory sharing. Figure 8 shows the different components' views of physical memory. At the top of the address space is memory reserved for seL4. This is the only memory the kernel ever accesses, and the kernel prevents user processes from reading or writing any kernel data structures within this region. The kernel treats all other memory as opaque.

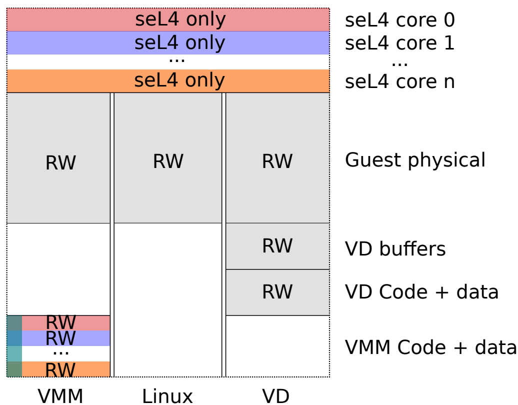

Figure 8. Memory map for the different components in the RapiLog system.

Furthermore, the multicore implementation of seL4 uses a multikernel approach (Baumann et al. 2009). This means that each core runs a separate instance of seL4, with its own private memory. Kernel instances do not communicate directly, instead user-level code can send IPIs and use a dedicated shared memory region for mailboxes. Hence, the "seL4 only" memory region of Figure 8 is further partitioned between kernel images, as indicated by the different colours. Our hardware platform supports hyper-threading. At present we treat hardware threads as separate cores (hence, the number of kernel images equals the number of cores times the number of hardware threads per core). Each thread appears to the SMP guest as a separate virtual CPU. The VMM executes locally on each virtual CPU, emulating virtualisation-sensitive instructions without reference to other virtual CPUs.

Compared to an approach which clusters a single kernel image across hardware threads and protects it by a single lock (von Tessin 2012), this approach potentially has a performance penalty. However, we did not find that to create problems (see Section 5.4), as in our use case there is very little inter-core communication that requires seL4 mechanisms.

The VMM and the VD both map all of the guest's "physical" memory into their own address space. This allows the VD to access the guest's I/O buffers directly, just like a real disk performing DMA. The VD and VMM each have their own private memory (the former containing the buffer for log data). These are separate to ensure isolation between the two components. The VMM's private memory is also partitioned into per-core instances, just like the kernel's memory, with the exception of a small amount (indicated by a green overlay in the figure) that needs to be shared for the state of the guest, emulated devices and inter-VMM mailboxes. The VD is not partitioned. While it has a server thread per core, locks are used to make its execution mostly single-threaded, greatly simplifying the implementation.

4.2 Details of operation

The VD driver in the guest OS communicates with the virtual disk through the VMM using the VMCALL instruction. The VMCALL is received by seL4 and sent to the VMM, which then passes the message to the VD through the use of synchronous IPC. Linux remains blocked on the VMCALL until the VMM receives a completion response (via IPC) from the VD. Linux will then be resumed with the response placed in one of its registers.

When the guest OS requests a write operation, the VD copies the data from the guest VM's memory into one of the buffers. The VD has access to all of Linux's memory, hence the VD driver can pass guest-physical addresses to the VD (and receive them back). If insufficient buffer space is available, the VD indicates a temporary resource-unavailability to the VD driver (which will then defer completion of any `fsync` operation). Once buffer space is freed up, the VD uses seL4 asynchronous notification to signal the VD driver, which then re-tries the I/O. Retrying is transparent to Linux, which only experiences occasional increased latencies of what normally seems to be blindingly fast "physical" I/O.

As the write requests to the VD do not come directly from the DBMS's logging service but from the Linux block layer, they are not necessarily sequential. While Linux uses the deadline I/O scheduler, the mainly sequential log append write requests are still intermixed with filesystem metadata write requests (e.g. when an index block is added to the log file's inode).

In order to avoid losing the advantage of mostly sequential writes, the VD uses multiple non-overlapping buffers to allow merging multiple sequential writes interspersed with other writes. The VD adds to a buffer only if this keeps the buffer's data sequential, else it picks a free buffer. If there are no clean buffers then we block the virtual writes until one is available. The number and size of the memory buffers is configurable as described in Section 5.6.

Normally the VD will only issue physical writes when at least one complete disk block is cached in a buffer. It prioritises large writes over short ones to optimise the use of available disk bandwidth. In order to avoid starving small writes, all buffers are flushed periodically (once a second).

This approach implies that the actual disk writes may be performed in a different order than intended by the file system. Re-ordering of writes does not result in incorrect behaviour, as long as we can guarantee that *all* buffered data will eventually reach the disk.

Additionally, the VD disables the log disk's write cache, to ensure that log data is persistent when the disk signals I/O completion. This is important in order to guarantee stability of the log data in the case of power failures (McKusick 2006; Nightingale et al. 2008), as discussed in Section 3.4.

The VD must also be able to service read requests. These happen as a result of Linux reading file-system meta data; DBMS X also reads back log data during a checkpoint. As the VD buffers act as a write-back cache, some care is required in providing the correct data in response to read requests, especially where only a portion of a read request is in the VD buffers.

4.3 Dependability

Our implementation of RapiLog seeks to minimise the opportunities for failure of the components supporting the durability property of the DBMS. Specifically, we use the formally-verified seL4 kernel, which is guaranteed never to crash (Klein et al. 2009). However, we note that the formal verification of seL4 (and hence the dependability guarantees) are strictly valid only for ARM processors. The x86 port used here therefore has at present somewhat lesser assurance of correctness. Also, we are using a multicore version of seL4, and its verification is work in progress.[3] However, even with these limitations, seL4 is a highly dependable platform, and the chance of any failure of seL4 is much smaller than that of, say, a failure in Linux that leads to data corruption, something the native DBMS setup does not protect against.

seL4 also has a proof of a core integrity property (Sewell et al. 2011). It essentially states that the kernel will never modify user memory (including memory used to hold kernel data structures that represent user objects, such as address spaces or threads) unless explicitly authorised through an appropriate write capability. This means that, with an appropriate distribution of capabilities to the various subsystems, the kernel is guaranteed to respect the memory map of Figure 8, and forces all system components to respect it—no rogue writes are possible.

In addition, we take care to minimise the code size as well as the privileges of critical software components. While the VMM and VD (like the microkernel) run in *VMX root mode*, only seL4 runs at kernel privilege (Ring 0) while the other components run at user privilege (Ring 3). In addition, the VMM and the VD are in separate address spaces.

Therefore, the correctness-critical VD is strongly isolated from the (fairly complex) VMM. The two components communicate via seL4 message-passing IPC (as well as shared memory buffers). This means that if the VMM crashes, the VD, which does not depend on virtualisation services, can continue to operate and flush any dirty buffer contents to disk before the system shuts down. The VMM code is no more critical than the Linux kernel code: while a bug could in theory corrupt data, in practice this is unlikely, as the VMM never touches any user data. Data corrupting bugs are more likely to be hidden in the millions of lines of Linux code. A simple *crash* of the VMM, like a crash of Linux, will not impact durability.

The rest of the system runs in a virtual machine (i.e. in VMX non-root mode). All required devices, except for the log disk but including the data disk and the network, are controlled by native drivers inside the Linux kernel (using pass-through I/O). We use the system's IOMMU to restrict those drivers to the parts of physical memory accessible to Linux (i.e. the part of RAM which is mapped to guest physical memory).

We strive to keep the design and implementation of the VD as simple as possible, in order to minimise the opportunity for introducing design flaws or implementation bugs. As advocated in Section 3.3, our low-level disk driver is a Termite-style synthesised driver, in order to further reduce the likelihood of bugs in correctness-critical parts of the system.

The driver was synthesised from formal specifications of the OS and the device interfaces (Ryzhyk et al. 2009). Unlike in this prior work, the device interface was not handwritten from data sheets, but is a Simics model of the device which was thoroughly tested against the behaviour of the real device.

While the driver itself is not formally verified, the above approach, combined with the simplicity of the driver, make critical bugs highly unlikely (particularly compared with the likelihood of Linux corrupting data).

Table 1 shows the size of the various components of RapiLog. The total durability-critical part (the VD including the disk driver) weighs in at only 1,619 LoC. Given the simplicity and small size of the VD code, it should be possible to

Component	Critical?	LoC
Virtual disk driver (Linux)	no	204
VMM	no	6,058
Virtual disk (w/o driver)	yes	1,174
Synthesised disk driver	yes	445

Table 1. Code sizes for the components in RapiLog

[3] Due to the multikernel approach, the verification essentially requires proving correct bootstrapping and that each instance respects the partitioning of kernel memory.

achieve high reliability of this component through traditional software-engineering methods (i.e. testing and code inspection). Even formal verification of the complete VD implementation is highly feasible (see Section 5.7 for details). This would make the VD truly bullet-proof, but we have not yet attempted this.

5. Evaluation

For evaluating the performance of the RapiLog prototype we used the above mentioned three database engines, two open source (MySQL and PostgreSQL) and a commercial DBMS (DBMS X). We measure the performance gains of RapiLog compared to baselines that have the same durability guarantees, as well as some unsafe and hardware-enhanced configurations. We also examine the sensitivity of RapiLog performance on the size of the VD's buffer, to ensure that it can be kept small enough to flush to disk safely in the case of power failure. We finally test the system's ability to survive power failures without database corruption.

5.1 Systems and platform

We host the database systems on Debian Linux, specifically the stable *squeeze* release based on Linux kernel version 2.6.32. We chose this version, rather than the most recent Linux kernel, because we believe that using a well-known, stable, distribution release and kernel will allow for better benchmark consistency and fewer bugs introduced from other modifications to the kernel (Harji et al. 2011). From the point of view of the RapiLog implementation, the actual Linux version is fairly irrelevant as the block driver API is very simple and unlikely to change much, but even if it did, the adaptation of our virtual driver would be trivial.

We recompile the kernel from source, including only those drivers that are necessary to provide the system's required functionality. We use MySQL 5.1 and PostgreSQL 8.4, the versions that are distributed with the stable Debian release.

Except where noted, RapiLog is configured with six buffers of 512 KiB each. Group commit is enabled except where explicitly mentioned, and no log multiplexing is used.

Our hardware platform is an IBM System X, model x3200 M3; its specifications are shown in Table 2. The system has two-high performance 10,000 RPM Serial ATA disk drives from Western Digital's Raptor product line. We use the *ext3* filesystem for all benchmarks.

Feature	Value
Processor model	Intel Xeon X3450
Cores	4
Hardware threads	2 per core
Clock speed	2.66 GHz
L1 D-cache	32 KiB
L2 cache	256 KiB
L3 cache	8 MiB
Main memory	4 GiB

Table 2. Characteristics of the benchmarking platform.

5.2 Configurations

We performed a thorough evaluation of the performance of the three DBMSes, each with three different setups. The **Native** setup consists of the DBMS running on top of an (unvirtualised) vanilla Linux installation, having the data and the transaction log on two separate disks. The **Virtualised** configuration runs Linux on top of seL4, with the DBMS running on top of Linux, again with separate disks for data and the transaction log. In this scenario, both disks are directly controlled by Linux drivers (I/O pass-through). By comparing with *Native*, this configuration lets us factor out the virtualisation overhead.

Finally the **RapiLog** setup extends the *Virtualised* setup, by performing asynchronous, yet durable, disk writes to the transaction log disk (as described in Section 3). For all experiments (unless otherwise stated), the write-cache of the transaction log disk is turned off to ensure durability. For the same reason, the *Native* and *Virtualised* configurations use synchronous writes to the transaction log.

For each of the 3 scenarios we test the performance of PostgreSQL, MySQL and DBMS X, based on the TPC-C Benchmark (TPCC). The dataset used is populated according to the benchmark specification with 10 Warehouses (yielding approx. 1 GiB of raw data and 1 GiB of indexed data). The OLTP style workload specified by the benchmark is generated by external clients, having no think time and performing synchronous requests to the DBMSes. The reported performance metric is the DBMS's throughput measured in successful TPC-C *NewOrder* transactions per minute.

We chose the TPC-C Benchmark because its many fast update transactions will put pressure on the transaction log. The TPC-C workload consists of a mix of 5 types of transactions of which 3 perform updates on the database (and account for approx. 92% of the total executed transactions).

For each measurement we allow a warm-up period, after which the throughput and request response times are recorded for the duration of the experiment. We tune all three database engines in order to achieve their best native performance given the workload and hardware described. Also, for minimising the impact of checkpointing data from the transaction log to the main data disk, we postpone this operation as much as possible. Frequent checkpointing has as main benefit a reduction in database recovery time in the case of a crash, but has no impact on the durability of the data. The actual number of checkpoints depend on operating conditions, and different databases use different checkpointing strategies. Moreover, these are not logged, so we cannot tell (without instrumenting the system) how many were performed, other than that they are relatively infrequent.

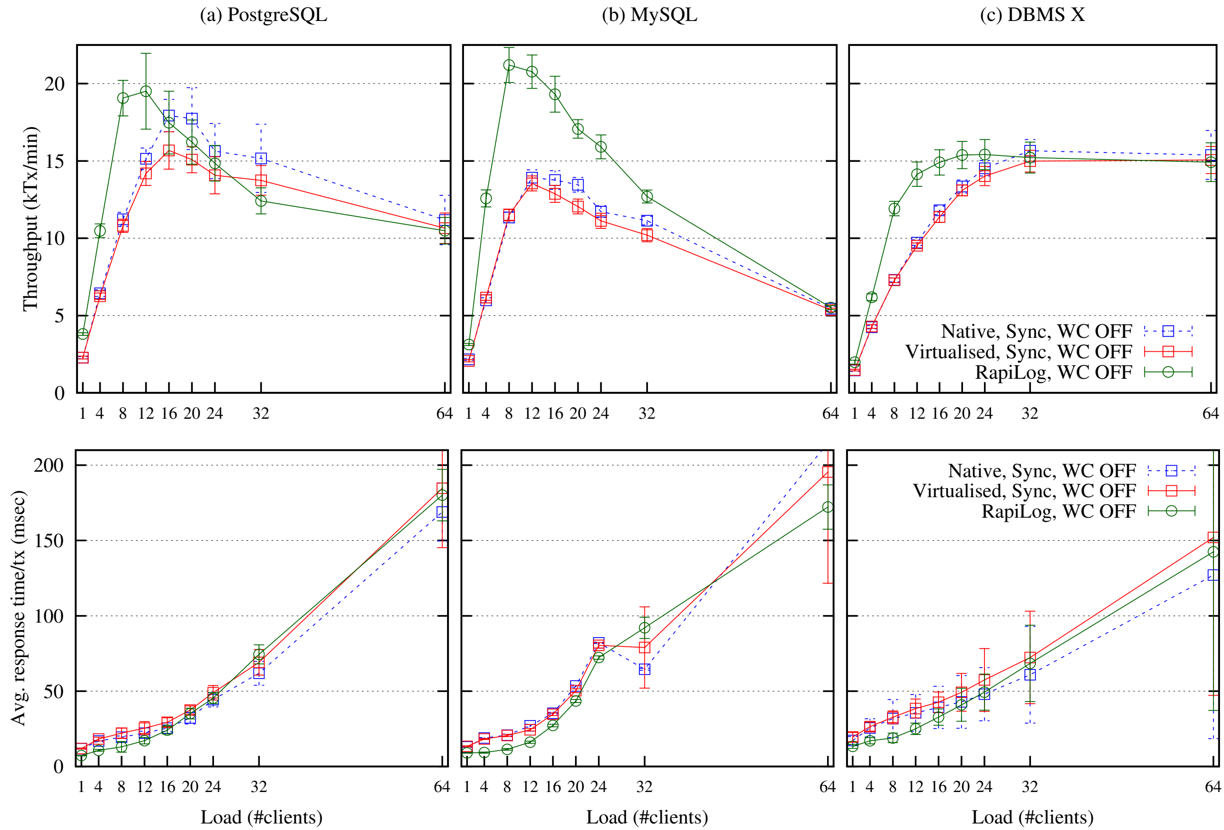

Figure 9. Performance of RapiLog compared to native and virtualised for PostgreSQL, MySQL and DBMS X.

5.3 Performance

Figure 9 shows a performance comparison of the Native, Virtualised and RapiLog setups for the three investigated databases. The top row of graphs show the throughput while the lower row shows average response times.

As discussed in Section 2.3, PostgreSQL hits an I/O bottleneck on the data disk. As long as the system is in its desirable, not overloaded state (which we refer to as *underload*), we see that RapiLog's throughput exceeds that of the Native and Virtualised setups; the performance improvement of RapiLog is highest at 8 clients (69% over Native, 76% over Virtualised) and averages 56%/62% over the underload range.[4] We also note that RapiLog hits the data disk I/O bottleneck faster—performance is I/O-limited at 8 clients compared to 12 for Native and Virtualised. RapiLog performance does not drop below that of Virtualised (within the significance indicated by the error bars).

Figure 9 (b) shows that for MySQL, RapiLog exceeds the performance of Native (and Virtualised) under all load conditions: the largest improvement is 110%/104% (Native/Virtualised) at 4 clients and the average improvement over the underload region is 80%/79%. The apparent performance improvement of Virtualised over Native in underload is tiny and well within the error bars.

Finally, Figure 9 (c) shows that for DBMS X, RapiLog similarly improves performance in underload. While the gain is somewhat less than in the other cases, it still reaches 63%/63% at 8 clients, and averages 52%/53% over the underloaded range. Performance under overload is (within the variance of the results) identical to Virtualised and about 5% below Native.

While the three databases exhibit different behaviour, two general observations can be made:

Firstly, RapiLog *never degrades performance* (beyond the virtualisation overhead).

Secondly, the RapiLog setup significantly improves the overall throughput for all three databases (by having lower latencies on performing the commits) where synchronous I/O causes a bottleneck. In our benchmarking scenario, this is the case in all non-overloaded workloads. The sweet spot in which databases are normally configured to run is at the high end of the non-overloaded regime, so these performance improvements correspond to realistic usage scenarios.

[4] These averages are obtained by comparing the areas under the curves from zero load to the onset of overload.

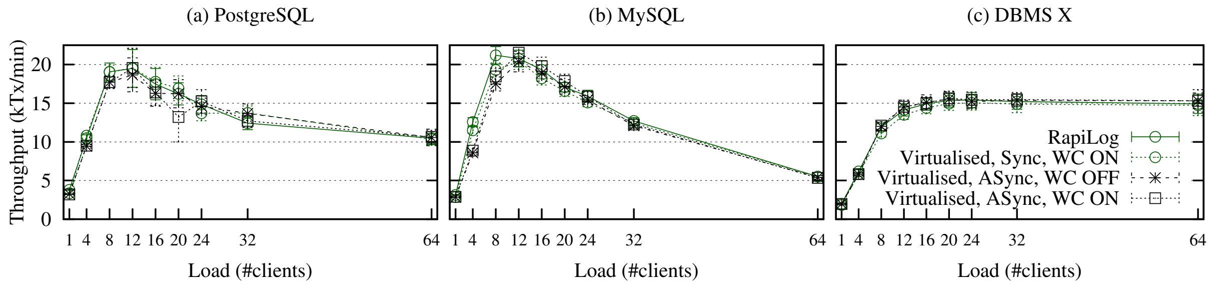

Figure 10. Effect of enabling unsafe asynchronous options for PostgreSQL, MySQL and DBMS X.

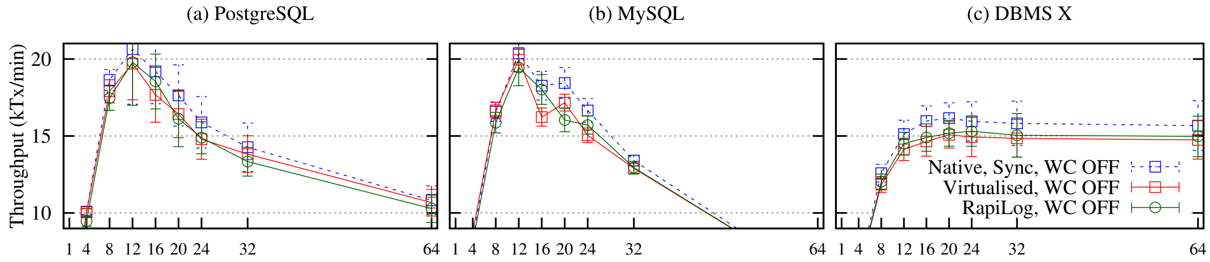

Figure 11. Performance of RapiLog compared to native on an SSD for PostgreSQL, MySQL and DBMS X.

5.4 Virtualisation overhead

On average, the throughput degradation of Virtualised over Native is 9% across the scenarios we measured, although the actual figure depends somewhat on the nature of the load. This figure seems comparable to commercial hypervisors.

To analyse further we profiled DMBS X with 32 clients and measured the amount of time spent outside of Linux, as shown in Table 3. Most of the CPU cores spend <1% of execution time outside of Linux, with the exception of one core, which Linux picked to handle network interrupts. Overall, about 1% of total CPU time (the average of the "%-age" row in Table 3) is spent outside the virtual machine. This means that there is little scope for reducing overheads by further optimising seL4 or the VMM. The remaining 8% overheads are likely hardware-imposed virtualisation overheads which are independent of the hypervisor, such as the (compared to Native) increased cost of handling TLB misses resulting from nested page tables.

5.5 Performance optimisations vs. RapiLog

In order to evaluate how RapiLog performs compared to the "corner-cutting" strategies, we ran several configurations of the Virtualised DBMS using asynchronous commit logging: (1) suppressing the use of `fsync`, (2) enabling the disk's write cache, and (3) the "belt & braces" configuration of both of these. All of these reduce the cost of logging, but do not provide durability in the case of a crash.

Core	0	1	2	3	4	5	6	7
Secs	5.3	35	5.4	5.3	5.3	4.3	5.4	5.3
%-age	0.6	4.2	0.6	0.6	0.6	0.5	0.6	0.6

Table 3. Execution time spent outside the Linux VM on each core for a 831 second benchmark.

Figure 10 shows the results. Within sampling error, the results are identical. This shows that RapiLog fully exploits the performance potential of asynchronous I/O (without sacrificing durability).

Another approach to reducing logging cost is using a faster persistent storage medium. Figure 11 shows the effect of using an SSD instead of a magnetic disk for persistent log storage. Within measurement accuracy, RapiLog performance equals Virtualised. Since the faster I/O device removes the bottleneck, RapiLog does not gain performance (but neither costs performance, other than the virtualisation overhead). This shows that RapiLog is a software-only alternative to more expensive hardware.

Finally, it would be interesting to see whether RapiLog can also remove the need for group commit. Our investigation of this point, shown in Figure 12, is inconclusive: for our benchmarking scenario, group commit has no performance

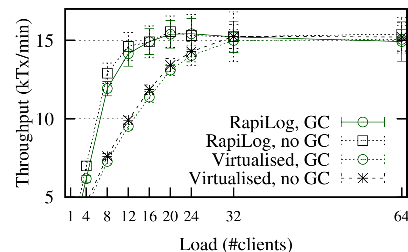

Figure 12. Performance impact of enabling group commit on DBMS X.

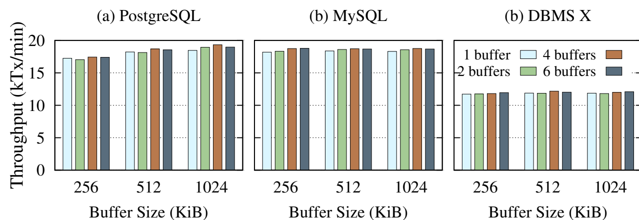

Figure 13. DBMS throughput as a function of buffer size and number.

effect (within sampling accuracy) in either the RapiLog nor the non-RapiLog case.

We can only speculate about the reasons, as we are not aware of any deep analysis done of the benefits of group commit on modern hardware. It is a feature designed decades ago, at a time when I/O bandwidth and main memory sizes were a fraction of what they are today. The potential gain of group commit is a complex function of the load, the amount of concurrent updates, and the I/O configuration. Therefore we are not surprised that for common database sizes and workloads, and most modern hardware, group commit no longer has a significant performance advantage.

5.6 Virtual disk buffer size

The perfectly matched curves of Figure 10 indicate that the buffers inside the VD were of sufficient size and did not create a bottleneck under any load. While it is obviously desirable to have sufficient buffer space to avoid contention, they should not be too big either, as else we risk not being able to flush them safely to disk if power is cut.

Figure 13 shows the results of a sensitivity analysis on the size and number of buffers. For each DBMS we chose the point where RapiLog shows the biggest performance gain over native; this is where RapiLog's buffering provides the biggest benefit, and thus should be most sensitive to buffer contention. We see that Postgres experiences a slight performance drop when reducing buffer size from 512 KiB to 256 KiB, while the other two systems were completely insensitive to buffer size in the investigated range. This justifies our choice of using 512 KiB buffers.

The figure also shows that there was almost no sensitivity to the total number of buffers in any of the DBMSes, so our choice of six buffers is, in hindsight, excessive, and could be reduced to increase the safety margin under power failure.

These results are in line with studies showing that fewer than 100 log pages of the log tail are actually active (Garcia-Molina and Salem 1992), which corresponds to about 400 KiB in our case.

5.7 Reliability

We measured the power window of our test machine, i.e., the time interval from cutting the external power supply until the machine stopped functioning, to be around 150 ms. This is in line with tests on other machines, where we had found window sizes of above 100 ms.

It is also in line with other recent research, which found power windows in the range of 10–400 ms (Narayanan and Hodson 2012). Note that the 10 ms minimum window found by Narayanan and Hodson is not directly comparable to the window relevant to RapiLog, for two reasons. Firstly, they measure the power window from the time the PowerGood signal (which indicates that the power rails are in spec) is deasserted by the hardware. The standard requires the power rails remain in spec (and PowerGood asserted) for at least 17 ms after the mains input is cut (Intel 2000), so their 10 ms window is really 27 ms for our purposes.

Furthermore, Narayanan and Hodson measure the power window under maximum power draw (and the 17 ms guaranteed by the standard also assume maximum power). In contrast, RapiLog explicitly *reduces* power draw to the bare minimum when detecting a power emergency, by suspending all system activities other than the (mostly sequential) writes to the log disk, including sleeping the CPU. Specifically, we observe that while our system's power draw can exceed 160 W, during emergency buffer flushing it is only around 80 W. The different power draw has a roughly linear effect on the size of the power window (< 80 vs. 150 ms). We can therefore estimate that even on the machine where Narayanan and Hodson observed a window of 10 ms, for our purposes it would be around 50 ms.

In order to test RapiLog's ability to maintain durability through power outages, we ran experiments where we cut power during database operation. For each such test we create a fresh database and proceed to insert rows with increasing key values. At a later time we cut power to the server, and record the last few transactions that were reported as successful. The VD logs the completion of emergency flushing of buffers to the console.

We cut power with the help of a USB-attached relay, which is controlled from a laptop. After turning power off, the laptop writes a character on a serial line connected to the system under test, thus triggering an interrupt, which is handled by the VD as a power emergency notification.

With our standard setup of 6 buffers of 512 KiB each, we find that the VD never needs to flush more than 3 buffers, which takes at most 20 ms, giving a healthy safety margin within the 150 ms power-out window.

The maximum value of three dirty buffers indicates once more that our six-buffer configuration is overkill. However, this result also indicates that the system is in a way self-limiting: it will not use more buffers than it really needs, so this over-provisioning seems harmless.

Upon restarting the machine, we firstly perform a filesystem check to allow the filesystem to recover from its log journal and repair any errors. We then start the DBMS. The DBMS detects the "unsafe" shutdown and proceeds to recover the missing transactions from the transaction log. We then compare the recovered database state with the expected

state, and determine whether the recovery was successful.

We performed 40 power-failure and recovery procedures for each DBMS, *and each time, the recovery of the committed state from the log was successful*. The effectiveness of the test became obvious when it failed on earlier versions of the VD. We traced these errors to subtle bugs, such as misplaced write barriers—more indication that concurrency is difficult to get right, and that formal verification should be used where possible.

6. Discussion

Optimising the database transaction log has been an ongoing task in the database community for many decades. While earlier work focussed on more sophisticated transaction processing (such as group commit) which inevitably increase complexity, most recent advancements rely on hardware improvements that can reduce the latency of write operations to the transaction log (e.g. solid-state drives).

Such hardware mechanisms add to cost (eg. solid-state storage is much more expensive than disk), and thus are not always applicable. Many real-world databases are therefore run in an unsafe mode of operation (using asynchronous logging).

RapiLog leverages advances in reliable operating systems, formal verification and device-driver synthesis, to enable the use of asynchronous I/O for logging without sacrificing durability. The above results show that RapiLog's performance is roughly at par with what can be achieved by using unsafe operation modes (asynchronous logging or on-disk write caches) or safe but expensive SSDs.

While we have demonstrated that RapiLog can be deployed with unmodified legacy systems, a potentially more interesting prospect is that it opens up the design space of transaction logs. RapiLog allows modularising durability management, and therefore enables factoring out complex durability-related code from databases into the operating system, simplifying configuration, management and deployment.

RapiLog is based on the assumption that the durability-critical parts of the system will not fail. In Section 4.3 we have discussed dependability threats, and explained how we mitigate these by keeping design and implementation simple. Even better would be to actually verify the VD implementation.

We roughly (and conservatively) estimate an effort of 2–3 person years. 3–6 months of this would be for developing an appropriate concurrency model (something which had been avoided in the verification of seL4 (Klein et al. 2009)). Another 3–6 months we estimate will be needed for a device model, and 1–2 person years for the actual verification of the implementation using current techniques.

There is no guarantee that the system can be verified as is. Conceivably, verification feasibility might require significant changes, which could possibly come at the expense of performance. We consider this unlikely: the virtual disk has been intentionally kept simple, and in our experience there is nothing in its implementation which would cause serious problems with verification. Furthermore, there is very little (beyond a simple `memcpy`) in the VD that is particularly performance critical, after all, it simulates a high-latency device.

Furthermore, we are exploring the use of code-and-proof synthesis from domain-specific languages for system components running on top of seL4. This has the potential of dramatically reducing the cost of developing verified components.

7. Conclusions and Future Work

We presented RapiLog, a prototype system exploring the possibility of replacing synchronous logging by a verified system enabling asynchronous log writes without compromising durability. We have demonstrated that this can lead to improved performance – of which even unmodified legacy systems can benefit.

RapiLog achieves this by buffering writes in memory and allowing the DBMS to continue transaction processing before those writes actually get recorded on disk. The underlying reliable microkernel ensures that log data gets safely flushed to disk even if the DBMS or its host OS crash, or external power is cut. Undeniably, this is just a first step but a very promising one.

As mentioned in Section 2.4, the present implementation of RapiLog cannot guarantee durability in the case of a fail-stop processor fault, while a traditional database setup handles this situation similar to an OS crash. We plan to extend RapiLog to handle a machine-check exception by rebooting the seL4 kernel, while taking care not to disturb any non-kernel memory. After rebooting, the kernel can then notify the VD (via a simulated power-down interrupt) of the need to flush the log buffer and shut down cleanly for a cold reboot.

RapiLog demonstrates that highly dependable software can be an alternative to using costly hardware or complex algorithms in defence against failures.

This observation also allows us, for example, to revisit the architecture of database engines by externalising modules, thereby opening many interesting opportunities for future research. For instance, logging and the memory buffer used in the storage manager of databases are tightly coupled. Similar ideas as those used in RapiLog could be used in the memory buffer to ensure that dirty pages are written to disk in the event of failures, thereby simplifying both recovery as well as logging. More long term, database collocation in a virtualised environment raises the opportunity to turn these externalised components into services for all instances running on the same physical machine, thereby achieving further optimisations and savings.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. The work of Tudor Salomie is partially funded by the Enterprise Computing Center of ETH Zurich. We would like to thank Leonid Ryzhyk, Adam Walker and John Keys for providing the Termite IDE driver, and Peter Chubb for building the power-out notification device and helping in many places. We would finally like to thank the anonymous EuroSys reviewers, and especially our shepherd Steven Hand, for insightful comments which helped to improve the paper.

References

E. Alkassar and M. A. Hillebrand. Formal functional verification of device drivers. In *VSTTE 2008*, pages 225–239, Toronto, Canada, Oct. 2008. Springer.

A. Baumann, P. Barham, P.-E. 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 *22nd SOSP*, Big Sky, MT, USA, Oct. 2009.

P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. E. Lowell. The Rio file cache: Surviving operating system crashes. In *ASPLOS*, pages 74–83, Oct. 1996.

D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, Boston, MA, USA, 1984.

H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4:509–516, 1992.

J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surveys*, 15:287–317, 1983.

A. S. Harji, P. A. Buhr, and T. Brecht. Our troubles with Linux and why you should care. In *2nd APSys*, pages 2:1–2:5, Shanghai, China, 2011.

G. Heiser, L. Ryzhyk, M. von Tessin, and A. Budzynowski. What if you could actually *Trust* your kernel? In *13th HotOS*, Napa, CA, USA, May 2011.

Intel. ATX/ATX12V power supply design guide. http://www.formfactors.org/developer%5Cspecs%5CATX_ATX12V_PS_1_1.pdf, 2000.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct. 2009.

D. B. Lomet. Persistent applications using generalized redo recovery. In *ICDE*, pages 154–163, 1998.

D. B. Lomet and M. R. Tuttle. A theory of redo recovery. In *SIGMOD Conference*, pages 397–406, 2003.

D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *16th SOSP*, St. Malo, France, Oct. 1997.

M. K. McKusick. Disks from the perspective of a file system. *USENIX ;login:*, 31(3):18–19, 2006.

Microsoft. Microsoft SQL Server documentation. http://msdn.microsoft.com/en-us/library/ms189275.aspx, May 2012.

C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17:94–162, 1992.

D. Narayanan and O. Hodson. Whole-system persistence. In *17th ASPLOS*, London, UK, Mar. 2012.

NICTA. seL4 download site. http://ertos.nicta.com.au/software/seL4/, Jan. 2011. Kernel binary and spec.

E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Trans. Comp. Syst.*, 26:6:2–26, 2008.

Oracle. Applications users guide. http://docs.oracle.com/cd/A85964_01/acrobat/115oaug.pdf, Sept. 2000.

Oracle. Database administrator's guide. http://docs.oracle.com/cd/B28359_01/server.111/b28310/onlineredo001.htm, Mar. 2008.

J. Ousterhout and F. Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.*, 23:11–28, Jan. 1989.

L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *22nd SOSP*, Big Sky, MT, USA, Oct. 2009.

T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug. 2011.

U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *5th EuroSys*, Paris, France, Apr. 2010.

M. Stonebraker. Operating system support for database management. *CACM*, 24:412–418, 1981.

TPCC. TPC-C homepage. http://www.tpc.org/tpcc/.

M. von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *2nd WS on Systems for Future Multi-core Architectures*, Berne, Switzerland, Apr. 2012.