OXenstored

An Efficient Hierarchical and Transactional Database using Functional Programming with Reference Cell Comparisons

Thomas Gazagnaire Vincent Hanquez

Citrix Systems
First Floor Building 101, Cambridge Science Park, Milton Road
Cambridge, CB4 0FY, United Kingdom
{thomas.gazagnaire,vincent.hanquez}@citrix.com

Abstract

We describe in this paper our implementation of the Xenstored service which is part of the XEN architecture. Xenstored maintains a hierarchical and transactional database, used for storing and managing configuration values.
We demonstrate in this paper that mixing functional data-structures together with reference cell comparison, which is a limited form of pointer comparison, is: (i) safe; and (ii) efficient. This demonstration is based, first, on an axiomatization of operations on the tree-like structure we used to represent the Xenstored database. From this axiomatization, we then derive an efficient algorithm for coalescing concurrent transactions modifying that structure. Finally, we experimentally compare the performance of our implementation, that we called OXenstored, and the *C* implementation of the Xenstored service distributed with the XEN hypervisor sources: the results show that OXenstored is much more efficient than its *C* counterpart.
As a direct result of this work, OXenstored will be included in future releases of XENSERVER, the virtualization product distributed by Citrix Systems, where it will replace the current implementation of the Xenstored service.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems – Transaction processing; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Language*]: Language Constructs and Features – Data types and structures

General Terms Algorithms, Design, Performance

Keywords Databases, Transactions, Concurrency, Prefix Trees

1. Introduction

XEN (Barham et al. 2003) is an open-source type 1 hypervisor, providing the ability to run multiple operating systems, called guests, concurrently on a single physical processor. Type 1 (or native, bare-metal) hypervisors are software systems that run directly on the host's hardware. They have been very popular architecture since the CP/CMS (Creasy 1981), developed at IBM in the 1960s. The XEN hypervisor is very popular and is used, for example, by the Amazon Elastic Compute Cloud project[1] which allows customers to rent computers on which they can run their own applications.

In a virtual architecture, each guest runs securely partitioned from others in a virtualized environment. Using a technique called para-virtualization, which involves modifying processors' administrative instructions into calls into the hypervisor, processors' efficiencies are close to native performance. When operating system modifications are not practical or impossible, the XEN hypervisor leverages the use of special instructions, called VMM instructions. These instructions give the ability to run the guest unmodified but trapping all administrative operations securely.

In the XEN architecture, a para-virtualized privileged guest called the "control domain" is in charge of all the I/O needs of the other guests. Consequently, all virtual guests' devices (disks, network interfaces) have to be handled at this level too. For example, each guest's virtual disk is associated with at least a process in the control domain. Subsequently, when a new guest starts, it is necessary that the corresponding processes have already been started in the control domain and configured correctly. In the XEN architecture, it is thus necessary to have a specific service in the control domain to exchange control and configuration data between guests. This service is called Xenstored and can be seen as a tuple space system, providing concurrent-safe access to a key-value association database. This service has originally been implemented in *C* and is distributed with the XEN hypervisor sources. We will refer to it as CXenstored.

This paper describes another implementation of the Xenstored requirements, done using *Objective Caml* (Leroy et al. 1996), and we will refer to it as OXenstored. This new implementation uses functional data-structures as well as reference cell comparison (Pitts and Stark 1993; Claessen and Sands 1999), which is a limited form of pointer comparison. OXenstored is a fifth of the size of CXenstored (around 2000 lines of code) and significantly improves the performance in several respects: particularly, it increases by a factor of 3 the number of guests per host (up to 160). Moreover, we formalized the new algorithms implemented by OXenstored and we proved that they are safe, that is concurrent accesses always leave the database in a consistent state. Finally,

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08. . . $5.00

[1] Part of the Amazon Web Services: http://aws.amazon.com/ec2/

as a result of this work, OXenstored will replace CXenstored in future releases of XENSERVER, the virtualization product commercialized by Citrix Systems[2] which is built on top of the XEN hypervisor. OXenstored sources can be found on:
`http://xenbits.xen.org/ext/xen-ocaml-tools.hg`

Consequently, we believe that these arguments demonstrate that (i) functional (and thus immutable) data-structures are the most intuitive and simple candidates for manipulating tree-like structured data; and (ii) reference cell comparisons can be used in a very safe way to obtain great performance as well as proven consistency.

This paper is organized as follows: Section 2 gives a high-level overview of the Xenstored service's architecture, as well as an example of how it interacts with the XEN framework. Section 3 gives an informal explanation of the algorithm used by OXenstored. Then, Section 4 formalizes the functional data-structure used in OXenstored for modeling the database and section 5 derives the notion of a transaction in this context. Then, we explain in Section 6 how, unlike CXenstored, OXenstored is able to coalesce concurrent transactions which are affecting distinct parts of the database. We conduct some experiments in Section 7 to validate our approach and show that, indeed, the simplistic way of handling transactions of CXenstored can makes the system live-lock in a very common situation (starting a lot of guests), as opposed to OXenstored. Finally, Section 8 compares our approach with other work focused on transactional databases and Section 9 discusses the results given in this paper.

2. Xenstored Design and Use-Case

We give in Section 2.1 an overview of the design of the Xenstored service and, in Section 2.2, an example use-case of how it interacts with the XEN hypervisor.

2.1 Overview of the Xenstored Design

The Xenstored service is a single-threaded daemon, running in the control domain. It is the central point of communication, as guests communicate to each others using it. Basically, it is a file-system-like database, where control data is hierarchically organized.

There is two different ways a client can connect and communicate with the Xenstored service:

- First, using the ring buffers. These are two circular buffers stored inside in the guests' memory: one for sending request to the Xenstored service and one for reading its replies.
- Second, using the Unix sockets. These are accessible only for processes running inside the control domain. Clients of the Xenstored service use the standard Unix read and write operations to access to the database.

From the client point of view, the Xenstored service offers a hierarchical key-value association database which has the following properties:

Structured key database Contents in the database are structured into nodes which are addressed by Unix filesystem-style paths. For example, the ith guest stores its virtual-disk configuration under the path `/local/domain/i/device/vbd/` in a XEN-based system. Each path corresponds to a node in the database and it can store a value even if it has some children.

[2] Citrix Systems, http://www.citrix.com.

Simple database operation The clients of the Xenstored service (which are either guests or are components running inside the control domain) have only a small set of atomic operations to set and get the contents from the database:

- `write`: create or modify a new path;
- `read`: get the value associated with a path;
- `mkdir`: create a new node in the database;
- `getperm`: get the permissions of a node;
- `setperm`: set the permissions of a node.

Notification The clients of the Xenstored service can ask to be notified of changes in a node. When a node is changed or created, all the guests watching this node or a parent of this node will be notified asynchronously that a specific path has been modified.

Permissions Each node has its own permissions. A permission is made up of an owner, which has all privileges on the node, and an access control list which specifies who is allowed to access or modify the node.

Quotas Unprivileged guests can be limited in the number of nodes they are authorized to create. This permits the Xenstored service's memory usage to stay at reasonable levels and this protects the control domain memory from malicious guests.

Transactions The clients of the Xenstored service have the ability to run a set of multiple operations modifying/reading the database in an atomic fashion. From the start to the end of the transaction, the Xenstored service will ensure that the database stays consistent. If a transaction cannot be made consistent, then the Xenstored service will reply to the client that the transaction needs to be retried. A client can have multiple opened transaction at the same time, and thus each transaction needs to be identified by a unique transaction ID – whenever a client sends a request to the Xenstored service, it will associate to its request such a transaction ID. A transaction ID of 0 means that the request is not associated to a transaction and will be executed directly.

These requirements have originally been implemented in CXenstored. However, in CXenstored, transactions are handled in a simplistic way. In a set of concurrent transactions, only one would be able to successfully complete, all the other transactions would have to retry from the beginning. This way of handling multiple concurrent transactions makes the mechanism really simple to provide consistency, however it also adversely affects the performance of the system: when the number of concurrent transactions is high, a slow or long transaction will be disadvantaged in an environment when only the first to finish can be completed successfully. In case a client of the Xenstored service is doing a small and fast transaction in a loop, this will cause a live-lock and some transactions of other clients will never be able to complete.

2.2 Example Use-Case

In this section we give a small example which describes the sequence of steps involved when starting a new guest. This sequence of steps involves three different clients of the Xenstored service:

- The Control domain's Kernel (CK). The kernel of the control domain is able to set up the virtual disk drivers and the virtual network drivers inside the control domain, for the new guests.
- The new Guest Kernel (GK). When running, the kernel of the guest will try to connect to its virtual disk and network drivers which should be running inside and exported by the control domain.

- The management Tool-Stack (TS). It is running inside the control domain and it receives direct orders from the user. Particularly, when a user wants to start a specific guest, the management tool-stack is in charge of initiating the guest start protocol.

Starting a guest can be summarized as follows. Lines are prefixed with the name of client doing the action (ie. (CK), (GK) or (TS)). Almost all of these steps should be done atomically (ie. using transactions) in order to not pollute the configuration database in case one of the three clients fails.

(TS) The management tool-stack queries the XEN hypervisor to create a fresh guest. At this point, the guest is paused. The XEN hypervisor responds to the management tool-stack by giving i, the ID of the created guest. Then, the management tool-stack creates a collection of paths `/local/domain/i/devices/...` in the database.

(TS) The management tool-stack notifies the control domain's kernel (whose guest ID is 0) that it wants to set up some kernel devices by atomically writing a collection of configuration keys in `/local/domain/0/backend/...`.

(CK) The control domain's kernel is watching for modifications on the path `/local/domain/0/backend`, and thus is notified by the Xenstored service that someone wants it to configure new drivers inside the control domain. Once this is done, the control domain's kernels write some special values in `/local/domain/0/backend` to notify the management tool-stack that the devices are ready.

(TS) The management tool-stack is notified that the devices are ready inside the control domain. It can now ask the XEN hypervisor to really start the guest.

(GK) When the guest kernel is booting, it will read the configuration values put in `/local/domain/i/devices/...` by the management tool-stack. Using these data, it will be able to configure the data-path of its devices correctly, through the drivers of the control domain's kernel.

3. Informal Description of the Algorithm

Basically, the database of OXenstored is modeled as an immutable prefix tree (Okasaki 1999). Each transaction is associated with a triplet (T_1, T_2, p), where T_1 is the root of the database just before the transaction starts, T_2 is the current local copy of the database with all updates made by the transaction up to that point in time, p is the path to the node furthest from the root of T_2 whose subtree contains all the updates made by the transaction up to that point. The transaction updates T_2 by substitution and copying all the nodes from the root of T_2 to the node where the substitution takes place. At the end of the transaction, the system tries to commit. At this point, the system checks if T_1 is still the root of the current database. If it is, it just commits by setting T_2 to be the current database. If it is not, it checks if the subtree at p in T_1 is same as the subtree at p in the current database. If it is, it means that no-one has yet touched the nodes touched by the transaction, so it just commits by substituting the subtree at p in the database by the subtree at p in T_2. Otherwise, someone must have touched the nodes touched by the transaction, and therefore the transaction must abort to ensure serialisability (Härder and Reuter 1983).

Let us now consider a small example to see how this algorithm works. Let us consider an initial database associating 0 to ε, 5 to a and 4 to b, and a transaction trying to associate 7 to b. This transaction is represented by the triplet (T_1, T_2, b), where T_1 and T_2 are two prefix trees sharing in memory the same node associated to the key a, as described in Figure 1. Moreover, let us assume that T, the prefix tree described in this figure, is the state of the database just before committing the transaction. When committing (T_1, T_2, b) to T, the system observes first that T_1 and T's roots are not in the same memory location. However, the subtree at b in T_1 is the same as the subtree at b in T. Thus, the new database state becomes the prefix tree T' in figure 1, which is the substitution of the subtree at b in T by the subtree at b in T_2. In the next sections, we formalize that algorithm and demonstrate that it works correctly.

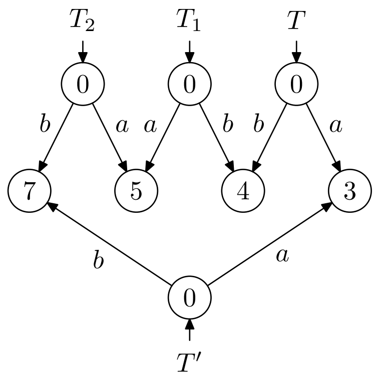


Figure 1. An example of how the algorithm implemented by OXenstored works.

4. Database Representation

In this section we introduce the data structures we decided to use for internally representing the OXenstored database, namely the *prefix trees*, or simply *tries* (Fredkin 1960), which are efficient structures for representing dictionaries. We only consider functional tries, ie. data-structures whose states are not mutable: an update operation on these structures creates a new structure and tries to share as much data as possible with the previous one. We first give in Section 4.1 some general definitions and properties of tries and we explain in Section 4.2 the more precise assumptions and choices we made for implementing a trie library in *Objective Caml*: this leads to an axiomatization of the trie data-structure library on which the following sections can rely on to prove the correctness of our transaction-coalescing algorithm.

4.1 Basic Materials

First, let us fix a finite set $\mathcal{K}$ of *keys* and let us consider the free monoid $\mathcal{K}^\star$, ie. the set of string over alphabet $\mathcal{K}$, defined as the infinite set of (possibly empty) key sequences $\bigcup_{n\in\mathbb{N}}\mathcal{K}^n$ and the composition law, having the empty sequence ε as neutral element and for which $uv = a_1 \dots a_n b_1 \dots b_m$ if $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$; and $u\varepsilon = \varepsilon u = u$. Elements of $\mathcal{K}^\star$ are called paths. A prefix of $u = a_1 \dots a_n$ is either ε, or a path $a_1 \dots a_k$ with $k \in \{1 \dots n\}$. A strict prefix of u is either ε or a path $a_1 \dots a_k$ with $k < n$. Moreover, the size of a path u, denoted by $|u|$, is the number of elements contained in the sequence; more formally, $|a_1 \dots a_n| = n$ and $|\varepsilon| = 0$. In OXenstored, a path is represented as a slash-separated list of names, as `/local/domain/0/device`. This path can be understood as the sequence of keys $a_1a_2a_3a_4$, where $a_1 =$ `local`, $a_2 =$ `domain`, $a_3 =$ `0` and $a_4 =$ `device`.

Second, let us fix a finite set $\mathcal{V}$ of *values*. A *trie* is a structure which partially maps paths to values. Thus, it can also be considered as a total function $T : \mathcal{K}^\star \to (\mathcal{V} \cup \{\bot\})$, where $\bot$ is a special symbol introduced to denote the fact that a path has no associated value.

Finally, the singleton trie associating a value x with the path ε and $\perp$ to anything else is denoted by $\{x\}$. The infinite collection of tries is denoted by $\mathbb{T}(\mathcal{K}, \mathcal{V})$ or simply by $\mathbb{T}$.

Tree Representation In practice, this hierarchical mapping can be used to optimize space utilization of a trie: indeed, it can then be implemented as a finite tree whose nodes and edges are labelled by values and keys respectively. Such a tree T can be decomposed into a structure $\langle x, \{a_i, T_i\}_{i \in \mathcal{I}}\rangle$, where $x \in \mathcal{V}$, $\mathcal{I} = \{1 \ldots n\}$ and for any $i \in \mathcal{I}$, $a_i \in \mathcal{K}$ and $T_i \in \mathbb{T}$ such that:

- $T(\varepsilon) = x$;
- For any $u = bv \in \mathcal{K}^\star$ such that $b \in \mathcal{K}$, $v \in \mathcal{K}^\star$, we have:
  - If $T(b) = \perp$ then there is no i such that $a_i = b$;
  - Otherwise $T(u) = T_i(v)$ where i is such that $a_i = b$.

Thus, for any path u in $\mathcal{K}^\star$ the value associated with u in T is either $T(u) \in \mathcal{V}$ if there is a node associated with the path u in the tree representing T, or $T(u) = \perp$ otherwise. Figure 3 shows such a trie T: paths `/vm/0`, `/vm/1` and `/vm/2` share the same prefix `/vm` and thus the values $T(\texttt{/vm/0})$, $T(\texttt{/vm/1})$ and $T(\texttt{/vm/2})$ are stored in the same subtree at `/vm` in T. For this trie, we also have $T(\texttt{/vm/3}) = \perp$.

We are now ready to define basic operations on tries. We consider in this paper two operations, namely the substitution and restriction.

Substitution First, the *substitution* operation consists to replace any subtree of the original trie by another subtree. More formally, the trie substitution can be defined as follows: given two tries T_1 and T_2 in $\mathbb{T}$ and a path u in $\mathcal{K}^\star$, the substitution of T_1 on path u by T_2, denoted by $T_1[u/T_2]$, is a new trie such that, for any path v and w in $\mathcal{K}^\star$, we have $T_1[u/T_2](uw) = T_2(w)$ and if u is not a prefix of v, then $T_1[u/T_2](v) = T_1(v)$. We can also extend these notations to define $T[u/x]$ where $T \in \mathbb{T}$ can be decomposed into $\langle y, \{a_i, T_i\}\rangle$, $u \in \mathcal{K}^\star$ and $x \in \mathcal{V}$, to be the substitution $T[u/\langle x, \{a_i, T_i\}\rangle]$.

Restriction Second, the *restriction* operation selects a specific subtree in the initial tree: given a trie T in $\mathbb{T}$ and a path u in $\mathcal{K}^\star$, the *restriction* of T to u, denoted by $T|u$, is a trie such that, for any path v in $\mathcal{K}^\star$, $(T|u)(v) = T(uv)$. The trie $T|u$ is also called a sub-trie of T. The restriction operation might also be seen as a *partial application* on tries. Figure 3 shows an example of trie restrictions: the trie whose nodes are labeled by b, d, e and f is a sub-trie of T, obtained its restriction to the path `/vm`, ie. it is $T|\texttt{/vm}$.

4.2 Axiomatization using Reference Cell Equality

When the question of implementing the substitution and restriction operators defined above in an efficient trie library arises, the programmer still has a lot of freedom: the given definitions do not explain directly how to express the substitution and restriction in terms of tree operations. Indeed, the above definitions hold for path/value associations only and nothing is said about the location of these values. In particular, nothing is specified about subtree sharing. However, every modern compiler of functional languages, such as *Objective Caml* (Leroy et al. 1996) or Scheme (Serrano 2000), enforces that multiple copies of an immutable structures share the same location in memory. is possible to design a trie library which enforces the sharing of subtrees as much as possible. In order to define more formally the notion of sharing, the *reference cell equality* (Pitts and Stark 1993; Claessen and Sands 1999), denoted by $\equiv$, has been introduced. This equality, also known as *physical equality*, is a limited form of pointer equality. It compares the location of values instead of the values themselves and thus can be used to observe value sharing: two values are shared if they have the same location, that is, if they are physically equal.

In light of this discussion, Figure 2 redefines the substitution and the restriction in order to enforce the sharing of values: rules (E1) and (E2) focus on the behavior of physical equality only; rules (S1), (S2), (S3) and (S4) focus on the substitution operator and its interactions with the restriction operator; finally, rules (R1), (R2) and (R3) focus on the restriction operator only. These definitions can be considered as axioms that any implementation of tries should satisfy and for which any formal reasoning can rely on.

(E1) $\forall T \in \mathbb{T}, T \equiv T$

(E2) $\forall T_1, T_2 \in \mathbb{T}$ and $\forall u, v \in \mathcal{K}^\star$,
if $T_1|u \equiv T_2|v$, then $T_1(u) = T_2(v)$

(S1) $\forall T_1, T_2 \in \mathbb{T}$ and $\forall u, v \in \mathcal{K}^\star$, $T_1[u/T_2]|uv \equiv T_2|v$

(S2) $\forall T_1, T_2 \in \mathbb{T}$ and $\forall u, v \in \mathcal{K}^\star$, $T_1[uv/T_2]|u \not\equiv T_1|u$

(S3) $\forall T_1, T_2 \in \mathbb{T}$ and $\forall u, v \in \mathcal{K}^\star$, if:
$\triangleright$ u is not a prefix of v and
$\triangleright$ v is not a prefix of u
then $T_1[v/T_2]|u \equiv T_1|u$

(S4) $\forall T_1, T_2 \in \mathbb{T}$ and $\forall u, v \in \mathcal{K}^\star$,
if $v \neq \varepsilon$, then $T_1[uv/T_2](u) = T_1(u)$

(R1) $\forall T \in \mathbb{T}, T|\varepsilon \equiv T$

(R2) $\forall T \in \mathbb{T}$ and $\forall u, v \in \mathcal{K}^\star$, $T|u|v \equiv T|uv$

(R3) $\forall T_1, T_2 \in \mathbb{T}$ and $\forall u \in \mathcal{K}^\star$,
if $T_1 \equiv T_2$, then $T_1|u \equiv T_2|u$

Figure 2. Axiomatization of the reference cell equality on tries, with respect to substitution and restriction operations.

Physical Equality More informally, first of all, the two first rules are related to the reference cell equality behavior only. Rule (E1) states that a given symbol always physically represents the same trie and rule (E2) states that reference cell equality implies usual structural equality: if the reference cells of two tries are identical, then they also associate the same values to the same keys.

Substitution Second, the next four rules are related to the substitution operator and its interactions with the restriction one. Rule (S1) states that sub-tries of a substituted trie are physically equivalent to sub-tries of the newly inserted trie. Rule (S2) states that nodes which are on the path of the substitution are never physically equivalent to the respective nodes in the initial trie: they correspond to newly allocated reference cells. Rule (S3) states that nodes which are not related to the substitution are not modified, ie. substitution is a local operation which enforces node sharing between tries produced by substitution. Finally, rule (S4) states that even though (S2) states that nodes which are on the path are newly allocated, the value they contain is preserved and is still equal to the value of the initial trie.

Restriction Finally, the last rules are related to the restriction operator. Rule (R1) states that the restriction of any trie to an empty path is the trie itself. Rule (R2) states that it is (physically) equivalent to restrict a trie twice with two given path that to restrict a trie by the composition of these two paths. Finally, rule (R3) states that if two tries are physically equal, then their restriction to the same path are also physically equal.

Tree Representation The tree representation introduced in Section 4.1 can be then reformulated using the axioms of Figure 2 to be the following: a tree T can be decomposed into a structure $\langle x, \{a_i, T_i\}_{i\in\mathcal{I}}\rangle$, where $\mathcal{I} = \{1 \dots n\}$, $x \in \mathcal{V}$ and for any $i \in \mathcal{I}$, $a_i \in \mathcal{K}$ and $T_i \in \mathbb{T}$ such that:

- $T(\varepsilon) = x$;
- There is no a_i such that $T(a_i) = \bot$;
- $T|a_i \equiv T_i$.

In this case, rule (R2) ensures that for any $u \in \mathcal{K}^\star$, $T|a_i u \equiv T|a_i|u$, that is $T|a_i u \equiv T_i|u$. Finally rule (E2) ensures that $T(a_i u) = T_i(u)$.

For example, Figure 3 gives the graphical representation of a trie T, which can be decomposed as follows:

$$\langle a, \{(\texttt{vm}, \langle b, \{(\texttt{0}, \langle d, \emptyset\rangle), (\texttt{1}, \langle e, \emptyset\rangle), (\texttt{2}, \langle f, \emptyset\rangle)\}\rangle), (\texttt{vss}, \langle c, \emptyset\rangle)\}\rangle$$

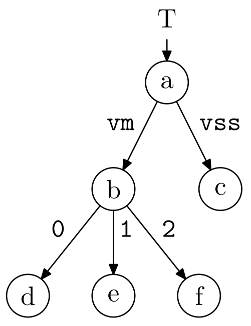


Figure 3. Example of a tree representation.

In this Figure, the values associated with the paths `/vm` and `/vm/1` are b and e respectively and $T|\texttt{vm}$ designates the sub-tree whose root is labeled by b.

5. Transactions

We stated that the Xenstored database can be represented as an immutable trie; this database can then be updated using the trie substitution operator as follows: if the trie $T \in \mathbb{T}$ is the current state of the database, then replacing the value associated with $u \in \mathcal{K}^\star$ by $x \in \mathcal{V}$ is done by replacing the current state of the database by the trie $T[u/x]$. However, Xenstored is a *transactional* database. That is, it is possible to ask for any sequences of access and modification to be done atomically.

First of all, in order to simplify the following definitions and results, we consider any reading operation as an identity substitution: for any trie T in $\mathbb{T}$ and path u in $\mathcal{K}^\star$, getting $T(u)$ does not modify T but when considering sequences of read/update operations it is necessary to remember that u had been read. So in this case, we write $T[u/(T|u)]$ as rule (S2) of Figure 2 states that in this case $T \not\equiv T[u/(T|u)]$. However, it is possible to extend the definitions and results we present in this paper to reading operations which do not update the database state.

We can now define transactions:

Definition 5.1 (transaction)**.** *A transaction is a sequence of substitutions. That is, a transaction σ belongs to $(\mathcal{K}^\star \times \mathbb{T})^\star$, ie. either σ is the empty sequence ε or $\sigma = [u_1/T_1] \dots [u_n/T_n]$, where for any $i \in \{1 \dots n\}$, $u_i \in \mathcal{K}^\star$ and $T_i \in \mathbb{T}$.*

Such a sequence can be applied to an initial trie $T \in \mathbb{T}$ in order to obtain a new trie $T' \in \mathbb{T}$, which is denoted by $T \xrightarrow{\sigma} T'$. This application consists of sequential application of each substitution $[u_i/T_i]$ for i from 1 to n, that is:

$$T' = ((\dots(T[u_1/T_1])\dots)[u_n/T_n])$$

For example, let us consider the following transaction:

1. Write $x \in \mathcal{V}$ in the path $u_1 \in \mathcal{K}^\star$;
2. Read the value associated with the path $u_2 \in \mathcal{K}^\star$;
3. Write $y \in \mathcal{V}$ in the path $u_3 \in \mathcal{K}^\star$.

Let us have the trie $T \in \mathbb{T}$ representing the current state of the database. Then the above transaction can be defined as:

$$\sigma = [u_1/(T_1|u_1)][u_2/(T_2|u_2)][u_3/(T_3|u_3)]$$

where T_1 is $T[u_1/x]$, T_2 is $T_1[u_2/T_1(u_2)]$ and T_3 is $T_2[u_3/y]$. Finally, T', the updated state of the database is such that $T \xrightarrow{\sigma} T'$.

Furthermore, the Xenstored requirements are that transactions can proceed concurrently. That is, multiple connections to the database can be opened and all of them can start independent transactions. As we consider immutable data-structures, this is not a problem: for each transaction started, the initial database is copied efficiently (as it is sufficient to copy only the location of the trie root, which is done in $O(1)$) and then modifications done by a transaction are applied directly to their own copy of the initial database.

However, ending (or "committing") a transaction is more complex. Indeed, the initial database might have been modified, since concurrent transactions or non-transactional events may have updated its state. It is thus necessary to carefully design what happens when a transaction ends.

First of all, Figure 4 describes the general algorithm which one has to solve when committing a transaction.

COMMIT ALGORITHM

Input:

- A trie $T_1 \in \mathbb{T}$
  (the initial database state)
- A modification sequence $\sigma_t \in (\mathcal{K}^\star \times \mathbb{T})^\star$
  (the transaction)
- A modification sequence $\sigma_1 \in (\mathcal{K}^\star \times \mathbb{T})^\star$
  (the concurrent modifications)

Output:

- A modification sequence $\sigma_2 \in (\mathcal{K}^\star \times \mathbb{T})^\star$
  (the consistent merging of σ_1 and σ_t)
- A value $r \in \{\mathsf{abort}, \mathsf{commit}\}$
  (the result of the merging)

Figure 4. The general commit algorithm.

Figure 5 illustrates this situation. In this figure, the trie T_1 is the initial state of the database, ie. its state just before the transaction starts; the trie T_t is the state of the database copy associated with the transaction, ie. is obtained by applying the transaction σ_t to T_1; the trie T_2 is the state of the database which might have been updated since the beginning of the transaction (σ_1 is empty if nothing has been executed concurrently with σ_t); furthermore, the trie T_3 is the state of the database after the modifications caused by σ_t have been taken into account to update T_2 accordingly; finally, the dashed lines between T_t and T_3 illustrates that this process is asymmetric: the T_i tries, for $i \in \{1, 2, 3\}$, are associated with visible states of the database, as T_t is an internal copy which carry modification information only. Thus, in case it is not possible to merge T_t with T_2, it is safe to completely discard the changes introduced by σ_t, that is to return $\sigma_2 = \varepsilon$ and $r = \mathsf{abort}$, to obtain $T_3 \equiv T_2$.

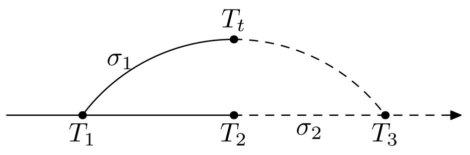


Figure 5. Relationship between tries used in the commit algorithm. Time goes from left to right.

Thus, there exist very simple algorithms which are able to merge changes introduced by transactions into the visible database state: the first one is the one which never commits anything, that is which always returns $\sigma_2 = \varepsilon$ and $r = \mathsf{abort}$. This behavior is correct, however it is not very useful in practice. Furthermore, CXenstored implements an algorithm which commits the changes only if T_1 has not been updated since the transaction has been started, ie. if and only if $T_1 \equiv T_2$. Otherwise, the transaction is aborted and retried after a short delay from the beginning. The hope is that nobody will update the database concurrently this time. In this case, T_3 is very simple to compute, as it is exactly T_t when the transaction is committed and T_2 otherwise. In practice, this simple algorithm is sufficient when transactions do not occur too often and it was implemented successfully by CXenstored. However, experiments show that in cases where the system is under load, this simple algorithm doesn't work any more as the transaction abort-and-retry mechanism live-locks (see Section 7). Thus we designed a better algorithm, able to *merge (or coalesce) concurrent transactions* and implemented it in OXenstored.

Regarding the the context of utilization of Xenstored, it is important to remark that transactions are localized and are closely related to the hierarchical structure of the database. Indeed, each guest has its own configuration values and the transactions it will create will access and modify only these values (and for security reasons, we do not want it to access or modify configuration values of other guests). For guests whose ID is i these configuration values are stored in specific sub-tries of `/local/domain/i`. Then accessing information about disk or network configuration can be done in accessing only the sub-tries `./device/vbd` and `./device/vif` respectively: it is then not necessary to block other transactions accessing different sub-tries to commit. In the following, we use that remark to coalesce concurrent transactions accessing distinct sub-tries of the database.

However, it is important to remember that we are still in a transactional model, that is some transactions will still eventually fail. So, there are still some corner-cases when, under load, the system will live-lock. However, in practice, Xenstored transactions often have a very specific shape (they are localized on some sub-tries) and thus the algorithm we give in the next section corrects this behavior and leads to more stable performance.

6. Transaction Coalescing

In the previous section, we defined transactions in terms of sequences of trie modifications. We explain in this section how to merge these transactions with the main state of the database. In order to properly explain how the coalescing algorithm we implemented in OXenstored works, we first introduce in Section 6.1 some basic definitions useful for dealing with trie modifications. In Section 6.2, we explain how optimizing a part of the coalescing algorithm, by incrementally updating what we will call the modification prefix of the transaction. Finally, we give the main OXenstored algorithm in Section 6.3 as well as its complexity in Section 6.4.

6.1 Main Results

The first of these definitions is about comparing the modifications done on two tries. More precisely, it is about locating the longest path which address sub-tries that are not physically equal in the two given tries. This longest path is called the *modification prefix* and can be more formally defined as the following:

Definition 6.1 (modification prefix)**.** *Let T_1 and T_t be two tries. The* modification prefix *of T_1 and T_t, denoted by $\pi(T_1, T_t)$, is an element u in $(\{\top\} \cup \mathcal{K}^\star)$ such that:*

- *If $T_1 \equiv T_t$ then $u = \top$;*
- *Otherwise, u is the longest path such that, for any $v \in \mathcal{K}^\star$:*
  - *If v is a strict prefix of u, then $T_1(v) = T_t(v)$;*
  - *If v is a prefix of u, then $T_1|v \not\equiv T_t|v$;*
  - *If $T_1|v \not\equiv T_t|v$ then either u is a prefix of v or v is a prefix of u.*

Hence, $\pi(T_1, T_t)$ is the longest path such that $T_t|\pi(T_1, T_t)$ is not a sub-trie of T_1 (and conversely, it the longest path such that $T_1|\pi(T_1, T_t)$ is not a sub-trie of T_t). Moreover, let us remark that in case $T_1(\varepsilon) \neq T_t(\varepsilon)$, then $\pi(T_1, T_t) = \varepsilon$, as there is no strict prefix of ε and any path has ε as prefix; moreover (E2) and (R1) state that $T_1(\varepsilon) \neq T_t(\varepsilon)$ implies that $T_1 \not\equiv T_t$. Furthermore, let us consider the trie T of Figure 5 and let us consider a new trie T_t obtained by applying the transaction $[uv/x][uw/y]$ to T, where $x, y \in \mathcal{V}$. In this case, one can check that $\pi(T, T_t)$ is exactly u.

The second of these definitions is about merging tries while enforcing a sub-trie sharing policy as much as possible. In order to understand the intuition of this definition, it is useful to consider the diagram shown in Figure 5. However, the following definition is more general and holds for any 3-tuple of tries:

Definition 6.2 (coalescing trie)**.** *Let T_1, T_2 and T_t be three tries. The trie T_3 is a* coalescing trie *of T_2 and T_t, relative to T_1, if, for any path v in $\mathcal{K}^\star$, it satisfies the following conditions:*

1. *If $T_1|v \equiv T_t|v$, then $T_3|v \equiv T_2|v$;*
2. *If v is not a strict prefix of $\pi(T_1, T_t)$ and $T_1|v \equiv T_2|v$, then $T_3|v \equiv T_t|v$;*
3. *In all cases, either $T_3(v) = T_t(v)$ or $T_3(v) = T_2(v)$.*

We are now ready to introduce the main result of this paper. The following theorem states how to compute the coalescing trie of three tries organized as in the diagram of Figure 5, ie. with an initial trie T_1 representing the initial state of the database, from which two distinct modification sequences lead to the two tries T_2 (the database's current state) and T_t (the local state associated with the current transaction). Basically, in the majority of cases, it is sufficient to substitute the database's current state by the sub-trie of T_t addressed by the modification prefix of T_1 and T_t. However, this theorem is not complete, that is there are some cases where building this coalesced trie is not possible. In this case, we can simply consider T_t as a trie related to a transaction and T_2 as the current state of the database, and as already discussed, in practice it is acceptable to discard the transaction trie T_t and let the database client retry the sequence of modifications.

Theorem 6.3 (coalescing tries). *Let T_1, T_2 and T_t be three tries and σ_t be a transaction such that $T_1 \xrightarrow{\sigma_t} T_t$. If $\pi(T_1, T_t) \neq \top$ and $T_1|\pi(T_1, T_t) \equiv T_2|\pi(T_1, T_t)$, then the coalescing trie of T_2 and T_t, relative to T_1, is:*

$$T_2\left[\pi(T_1, T_t)\ /\ (T_t|\pi(T_1, T_t))\right]$$

Proof. Let us fix $u = \pi(T_1, T_t)$. We assume that $u \neq \top$ and thus we can fix T_3 to be $T_2[u/(T_t|u)]$. Now, we want to check that the three assertions of Definition 6.2 holds for T_3.

Before starting the core of the proof, let us show a useful result. From the definition of T_3, we have:

$$\begin{array}{lcll} T_3|u & \equiv & T_2[u/(T_t|u)]|u & \text{(using (R3))} \\ & \equiv & (T_t|u)|\varepsilon & \text{(using (S1))} \\ & \equiv & T_t|u & \text{(using (R2))} \end{array}$$

Thus, we can fix, within the scope of this proof, the following assumptions:

$$\begin{array}{ll} \text{(A1)} & T_3 \equiv T_2[u/(T_t|u)] \\ \text{(A2)} & T_1|u \equiv T_2|u \\ \text{(A3)} & T_3|u \equiv T_t|u \end{array}$$

We are now ready to prove Theorem 6.3. v, w are in $\mathcal{K}^\star$.

1. Following the structure of Definition 6.2, we first need to prove that $T_1|v \equiv T_t|v$ implies that $T_3|v \equiv T_2|v$. To do this, let us consider the following three possible cases: (a) v is a prefix of u; (b) u is a prefix of v and (c) neither u is a prefix of v nor v is a prefix of u.

   (a) If $u = vw$:
   $$\begin{array}{lcll} T_1|v \equiv T_t|v & \Rightarrow & (T_1|v)|w \equiv (T_t|v)|w & \text{(using (R3))} \\ & \Rightarrow & T_1|u \equiv T_t|u & \text{(using (R2))} \end{array}$$
   However, as u is a valid prefix of u, Definition 6.1 states that $T_1|u \not\equiv T_t|u$. Contradiction.

   (b) If $v = uw$:
   Using (A1), we have $T_3 \equiv T_2[u/(T_t|u)]$. Then, we have:
   $$\begin{array}{lcll} T_3|v & \equiv & T_2[u/(T_t|u)]|uw & \text{(using (R3))} \\ & \equiv & (T_t|u)|w & \text{(using (S1))} \\ & \equiv & T_t|v & \text{(using (R2))} \end{array}$$
   Thus, $T_1|v \equiv T_t|v$ implies that $T_1|v \equiv T_3|v$.
   Then, let us develop (A2):
   $$\begin{array}{lcll} T_1|u \equiv T_2|u & \Rightarrow & (T_1|u)|w \equiv (T_2|u)|w & \text{(using (R3))} \\ & \Rightarrow & T_1|v \equiv T_2|v & \text{(using (R2))} \end{array}$$
   Thus, $T_1|v \equiv T_t|v$ implies that $T_3|v \equiv T_2|v$.

   (c) If neither u is a prefix of v nor v is a prefix of u:
   Let us start from (A1):
   $$\begin{array}{lcll} T_3|v & \equiv & T_2[u/(T_t|u)]|v & \text{(using (A1))} \\ & \equiv & T_2|v & \text{(using (S3))} \end{array}$$

   Thus, we showed that, in every case, $T_1|v \equiv T_t|v$ implies that $T_3|v \equiv T_2|v$.

2. Second, let us prove that if v is not a prefix of u and $T_1|v \equiv T_2|v$, then $T_3|v \equiv T_t|v$. To do this, let us consider the following two possible cases: (a) u is a prefix of v and (b) neither u is a prefix of v nor v is a prefix of u.

   (a) If $v = uw$:
   Let us start from (A3):
   $$\begin{array}{lcll} T_3|u \equiv T_t|u & \Rightarrow & (T_3|u)|w \equiv (T_t|u)|w & \text{(using (R3))} \\ & \Rightarrow & T_3|v \equiv T_t|v & \text{(using (R2))} \end{array}$$

   (b) If neither u is a prefix of v nor v is a prefix of u:
   Using Definition 6.1, we have $T_1|v \equiv T_t|v$.
   Then, we start from (A1) to obtain:
   $$\begin{array}{lcll} T_3|v & \equiv & T_2[u/(T_t|u)]|v & \text{(using (R3))} \\ & \equiv & T_2|v & \text{(using (S3))} \end{array}$$
   Thus, if $T_1|v \equiv T_2|v$, then $T_3|v \equiv T_1|v$ and thus, $T_3|v \equiv T_t|v$.

   Thus, we showed that, in every case, if v is not a prefix of u and $T_1|v \equiv T_2|v$, then $T_3|v \equiv T_t|v$.

3. Finally, let us prove that all cases, $T_3(v) = T_t(v)$ or $T_3(v) = T_2(v)$. To do this, let us consider the following three possible cases: (a) v is a prefix of u; (b) u is a prefix of v and (c) neither u is a prefix of v nor v is a prefix of u.

   (a) If $u = vw$:
   (A3) states that $T_3|v \equiv T_2[u/(T_t|u)]|v$. Then using (S4), we obtain that $T_3(v) = T_1(v)$.

   (b) If $v = uw$:
   Let us start from (A3):
   $$\begin{array}{lcll} T_3|v \equiv T_2[u/(T_t|u)]|v & \Rightarrow & T_3|v \equiv T_t|v & (S3) \\ & \Rightarrow & T_3(v) = T_t(v) & (E2) \end{array}$$

   (c) If neither u is a prefix of v nor v is a prefix of u:
   Let us start from (A3):
   $$\begin{array}{lcll} T_3|v \equiv T_2[u/(T_t|u)]|v & \Rightarrow & T_3|v \equiv T_2|v & (S3) \\ & \Rightarrow & T_3(v) = T_2(v) & (E2) \end{array}$$

   Hence, for all cases, we showed that either $T_3(v) = T_t(v)$ or $T_3(v) = T_2(v)$. □

6.2 Computing the Modification Prefix

In the last section, we explained how to properly coalesce transactions: it suffices to substitute the database's current state by the transaction state on the modification prefix of the transaction trie and initial database state. However, computing the modification prefix of two tries can be costly if we do not have additional information. Fortunately, the modification sequence of the transaction can be used to efficiently compute that modification prefix:

Lemma 6.4. *T_1 and T_t are two tries and $\sigma_t = [u_1/\widehat{T}_1] \dots [u_n/\widehat{T}_n]$ be a non-empty transaction such that $T_1 \xrightarrow{\sigma_t} T_t$. Then $\pi(T_1, T_t)$, the modification prefix of T_1 and T_t, is exactly the longest path which is a common prefix of every u_i, for $i \in \{1 \dots n\}$.*

Proof. Let us prove Lemma 6.4 by induction on the size of σ_t.

(i) Let us show the first induction step. Let us fix $\sigma_t = [u_1/\widehat{T}_1]$, that is, $T_t \equiv T_1[u_1/\widehat{T}_1]$ and let us prove that the longest common prefix of u_1 is the modification prefix of T_1 and T_t, that is $u_1 = \pi(T_1, T_t)$. According to Definition 6.1, we have to show three implications:

- First, we have to prove that, for any $v \in \mathcal{K}^\star$, if v is a strict prefix of u_1, then $T_1(v) = T_t(v)$: if v is a strict prefix of u_1, that is $u_1 = vw$ with $w \neq \varepsilon$, then (S4) states that $T_1[u_1/\widehat{T}_1](v) = T_1(v)$, that is $T_t(v) = T_1(v)$;
- Second, we have to prove that, for any $v \in \mathcal{K}^\star$, if v is a prefix of u_1, then $T_1|v \not\equiv T_t|v$: if v is a prefix of u_1, that is $u_1 = vw$, then (S2) states that $T_1[u_1/\widehat{T}_1]|v \not\equiv T_1|v$, that is $T_t|v \not\equiv T_1|v$;

- Finally, we have to prove that, for any $v \in \mathcal{K}^\star$, if $T_1|v \not\equiv T_t|v$, then either u_1 is a prefix of v or v is a prefix of u_1: if $T_1|v \not\equiv T_t|v$, that is $T_1|v \not\equiv T_1[u_1/\widehat{T}_1]|v$, then (S3) states that either u is a prefix of v or v is a prefix of u.

(ii) Let us then complete the induction process. Let us fix $\sigma_t = \sigma[u_n/\widehat{T}_n]$, with σ a non-empty transaction of size $n-1$ and let us consider the trie T such that $T_1 \xrightarrow{\sigma} T$. Let us then have $\pi = \pi(T_1, T)$. The induction hypothesis gives us that π is also the longest common prefix of $\{u_1, \dots, u_{n-1}\}$. Hence, for every $v \in \mathcal{K}^\star$:

   (a) If v is a strict prefix of π, then $T_1(v) = T(v)$;
   (b) If v is a prefix of π then $T_1|v \equiv T|v$;
   (c) If $T_1|v \equiv T|v$ then either v is a prefix of π or π is a prefix of v.

   Let u be the longest common prefix of $\{u_1, \dots, u_n\}$ and let us show that u is also the modification prefix of T_1 and $T[u_n/\widehat{T}_n]$. According to Definition 6.1, we have to show three implications:

   - First, we have to prove that, for any $v \in \mathcal{K}^\star$, if v is a strict prefix of u, then $T_1(v) = T_t(v)$: if v is a strict prefix of u, we have $u = vw$ with $w \neq \varepsilon$, and thus we have $u_n = vw$ with $w \neq \varepsilon$. Then (S4) states that $T[u_n/\widehat{T}_n](v) = T(v)$. Using (a), we obtain that $T_t(v) = T_1(v)$, with v being the longest common prefix of $\{u_1, \dots, u_n\}$;
   - Second, we have to prove that, for any $v \in \mathcal{K}^\star$, if v is a prefix of u, then $T_1|v \equiv T_t|v$: if v is a prefix of u, we have $u = vw$, for $w \in \mathcal{K}^\star$, thus we have $u_n = vw$. We can then use (S2) to obtain that $T[u_n/\widehat{T}_n]|v \not\equiv T_1|v$. Using (b), we obtain that $T_t|v \equiv T_1|v$, with v being the longest common prefix of $\{u_1, \dots, u_n\}$;
   - Finally, we have to prove that, for any $v \in \mathcal{K}^\star$, if $T_1|v \not\equiv T_t|v$, then either u is a prefix of v or v is a prefix of u: if $T_1|v \not\equiv T_t|v$ then it is due either to $T_1|v \not\equiv T|v$ or to $T|v \not\equiv T_t$. In the first case, (c) states that either v is a prefix of π or π is a prefix of v; in the second one (S3) states that either v is a prefix of u_n or u_n is a prefix of v. Thus, if p is the longest common prefix of π and u_n, that is the longest common prefix of $\{u_1, \dots, u_n\}$, then we have either v is a prefix of p or p is a prefix of v;

Thus we showed that Lemma 6.4 is valid for any size of σ_t. □

It is then straightforward to derive from Lemma 6.4 an incremental algorithm to compute the modification prefix of tries associated with a transaction: indeed, at each step of the sequence, it suffices to take the longest common prefix between the path currently modified by the current step and the modification prefix already computed from the beginning of the transaction.

6.3 Algorithms

OXenstored is able to properly *coalesce* unrelated transactions, ie. transactions which modify disjoint subtrees of the current store. To do this efficiently, it exploits the functional tree representation of the database. More precisely, consider a transaction $\sigma = [u_1/T_1] \dots [u_n/T_n]$. This transaction is executed incrementally so we can also consider a sequence of time t_k, for $k \in \{0 \dots n\}$, where t_0 corresponds to a time just before the transaction starts, ie. to the sub-sequence $\sigma_0 = \varepsilon$ of σ and each t_k corresponds to a time when only the sub-sequence $\sigma_k = [u_1/T_1] \dots [u_k/T_k]$ of σ has been executed. We can now define, for any $k \in \{0 \dots n\}$, the state of a transaction at the time t_k as a structure $\langle T^k, T_t^k, \pi^k \rangle$, where:

- $T^k \in \mathbb{T}$ is a snapshot of the state of the database just before the transaction started.
- $T_t^k \in \mathbb{T}$ is a local trie attached to the transaction and where the modification it contains are done, while letting the main database state unchanged until the transaction is committed. Its value is such that $T^0 \xrightarrow{\sigma_k} T_t^k$;
- $\pi^k \in (\{\top\} \cup \mathcal{K}^\star)$ is either $\top$ if no modifications yet been executed (ie. $k = 0$) or u if u the modification prefix associated with T^k and T_t^k (ie. $\pi(T^k, T_t^k)$), that is, it is the longest such that $T_t^k|\pi^k$ is not a sub-trie of T^k.

As already stated in Section 2.1, OXenstored gives a unique identifier to each started transaction. This identifier is created when a client sends to OXenstored a starting request for a new transaction; this identifier is associated internally with the structure $\langle T, T, \top \rangle$, where T is the current state of the database. This identifier is also sent back to the client; the client can then put this identifier in the header of packets it will send later in order to update the state of this specific transaction. Eventually, it can also notify OXenstored that it wants the transaction to commit, ie. to push the changes introduced by the transaction into the current state of the database.

In the following, we give the algorithms used by OXenstored to update the structure associated with a transaction and to commit the changes induced by a transaction into the current database state. We assume here that (i) a client already started a transaction σ_k, whose associated structure is $\langle T^k, T_t^k, \pi^k \rangle$; (ii) the client has been sending the requests to either update or commit a transaction; and (iii) OXenstored has already decoded the packet header of that request and the associated transaction structure is exactly $\langle T^k, T_t^k, \pi^k \rangle$.

Updating a Transaction First of all, let us detail how to update the structure associated with a transaction. Let us assume that OXenstored decoded the packet content and found that the clients want to substitute the trie T_{k+1} on path u_{k+1}. OXenstored has to compute the new transaction structure $\langle T^{k+1}, T_t^{k+1}, \pi^{k+1} \rangle$. We have:

- For any $k \in \{1 \dots n\}$, T^k is identical, as it is a snapshot of the state just before the beginning of the transaction;
- Definition 5.1 states that a transaction is updated by applying to its local state the substitution $[u_{k+1}/T_{k+1}]$;
- Lemma 6.4 shows that the modification prefix can be computed online: indeed it is sufficient to compute incrementally the longest prefix of π^k and u_{k+1}.

```
**Input**: $\langle T^k, T_t^k, \pi^k \rangle$ and the substitution $[u_{k+1}/T_{k+1}]$
**Output**: the new transaction structure $\langle T^{k+1}, T_t^{k+1}, \pi^{k+1} \rangle$

$T^{k+1} \leftarrow T^k$;
$T_t^{k+1} \leftarrow T_t^k[u_{k+1}/T_{k+1}]$;
$\pi^{k+1} \leftarrow$ longest-common-prefix($\pi^k$, $u_{k+1}$);
**return** $\langle T^{k+1}, T_t^{k+1}, \pi^{k+1} \rangle$
```

Algorithm 1: Updating a transaction.

These lead directly to Algorithm 1, that explains how to compute $\langle T^{k+1}, T_t^{k+1}, \pi^{k+1} \rangle$ from $\langle T^k, T_t^k, \pi^k \rangle$ and $[u_{k+1}/T_{k+1}]$, for any $k \in \{1 \dots (k-1)\}$. Note that this algorithm uses the function `longest-common-prefix(u,v)` which returns either the longest common prefix of u and v if $u, v \in \mathcal{K}^\star$ or returns v if $u = \top$ (and u if $v = \top$).

Committing a Transaction Second, let us detail how to commit the structure associated with a transaction into the current state of the database, in coalescing concurrent transactions when possible. Let us assume that OXenstored decoded a packet whose header contains an identifier internally associated with the structure $\langle T^k, T^k_t, \pi^k\rangle$ and whose content contains a commit order. OXenstored has to compute the new state of the database, which should share, as much as possible, nodes from the current database state T and from the transaction local state T^k_t; more precisely, we want to ensure that this new database state is exactly the coalescing trie of T and T^k_t, relative to T^k (see Definition 6.2). Basically, there are three cases:

- The first case is when no modifications are done concurrently with the current transaction on the entire database. This case can be checked easily. Indeed, rule (S2) states that, in case a substitution is done on between T and T^k, then there exists $u \in \mathcal{K}^\star$ such that $T|u \not\equiv T^k$. Finally, rule (R3) states that $T \not\equiv T^k$. When this case is detected, it is then safe to replace the database's current state by the local state of the transaction.
- The second case is when no modifications are done concurrently with the current transaction on the sub-trie corresponding to the scope of that transaction. Being able to detect this case efficiently is the main improvement from CXenstored to OXenstored. However, in our settings, this is relatively easy. Indeed, Definition 6.1 states that π^k is the longest path such that $T^k_t|\pi$ is not a sub-trie of T^k, that is $T^k_t|\pi$ is the biggest sub-trie modified by the current transaction. Thus, to detect if no modifications are done concurrently with the current transaction, it suffices to check that the corresponding sub-trie in the database's current state has not been modified, ie. that $T_t|\pi \equiv T^k_t|\pi$ (as stated by rules (S2) and (R3), as in the last bullet). When this case is detected, Theorem 6.3 states it is safe to replace the sub-trie of the database's current state by the corresponding sub-trie of the transaction's local state.
- The last case relates to aborting the transaction if it is not possible to commit it. In this case, the database's current state remains unchanged and the client is notified that it has to retry the transaction.

These lead to Algorithm 2.

```
Input: the current database state $T$ and $\langle T^k, T^k_t, \pi^k\rangle$
Output: the new database state and the transaction status

if $T \equiv T^k$ then
    /* if the database has not been updated   */
    return ($T^k_t$, commit);
else if $T|\pi^k \equiv T^k|\pi^k$ then
    /* if the sub-trie has not been updated   */
    $T' \leftarrow T[\pi^k/(T^k_t|\pi^k)]$;
    return ($T'$, commit);
else
    /* otherwise, abort                       */
    return ($T$, abort);
end
```

Algorithm 2: Committing a transaction.

Extension In the extended version of our implementation, where we do not consider that reading operations update the state of the database, the algorithm remains more or less the same. However, instead of keeping a unique modification prefix π^k for each transaction, we keep two of them: one for read operations, π^k_R, and for one write operations, π^k_W. Then, for deciding if the sub-trie of the transaction have been concurrently updated in the database's current state, we check that $T|\pi^k_R \equiv T^k|\pi^k_R$ and $T|\pi^k_W \equiv T^k|\pi^k_W$ stands. If this is the case, we only update the sub-trie corresponding to writing operations, ie. we set $T' \leftarrow T[\pi^k_W]/(T^k_t|\pi^k)$. This could be helpful to reduce the conflict rate between concurrent transactions which read and write in distinct sub-tries and dramatically increase the efficiency of the coalescing in case a transaction only performs read operations.

6.4 Complexity

In order to find the complexities of the algorithms, we can first derive the complexity of the trie operations from the axioms of Figure 2 and the tree representation detailed in Section 4.2.

First of all, using the tree representation of tries and assuming a linear complexity for accessing children of a node it is then straightforward to obtain that, given tries $T, T' \in \mathbb{T}(\mathcal{K}, \mathcal{V})$ and a path $u \in \mathcal{K}^\star$, the complexity of computing the restriction $T|u$ and the substitution $T[u/T']$ is $O(|u||\mathcal{K}|)$ (thus, independent of the size of T or of T'). Moreover, the complexity of computing the physical equality $T \equiv T'$ is in $O(1)$.

Thus, the complexity of Algorithm 1 is in $O(|u_{k+1}||\mathcal{K}|)$, where $|u^k|$ is the size of the path which is modified and the complexity of Algorithm 2 is in $O(|\pi^k||\mathcal{K}|)$, where $|\pi^k|$ is the size of the longest common prefix of any path which are modified. Thus, the complexity of these algorithms does not depend at all of the size of the current database, which gives very stable performance. Moreover, in practice, the number of keys $\mathcal{K}$ and the path size are very rarely over 10, which can already lead to databases with 10^{10} entries, while keeping a good level of efficiency for modification operations.

7. Evaluation

In the previous sections, we explained how the OXenstored algorithm for coalescing transactions works and we argued that, although transactions can still be aborted, the rate of committed transactions in a practice should be very high. In this section, we validate this statement experimentally, by comparing the performance of CXenstored and OXenstored. Not surprisingly, the results we obtained show that OXenstored scales much better than CXenstored: indeed CXenstored exhibits a live-lock in a very common situation (sequentially starting as many guests as possible), unlike OXenstored.

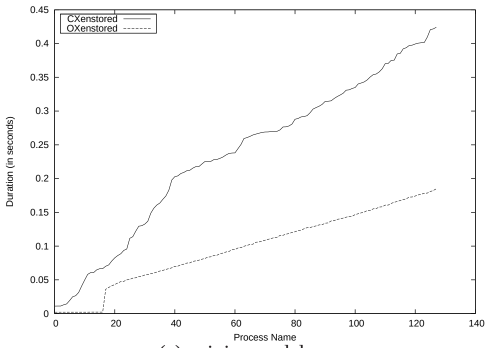
More precisely, the tests we ran are the following: first in Section 7.1, we measured the performance under load of CXenstored and OXenstored in an isolated context, by creating random transactions and without interacting with the XEN hypervisor. Then, we measured the performance of CXenstored and OXenstored when they are interacting with the XEN hypervisor, with and without load, in Section 7.2 and Section 7.3 respectively.

7.1 Performance under Load

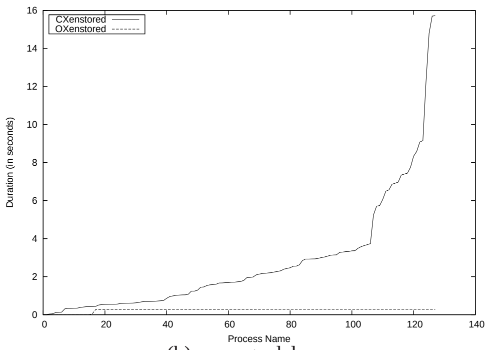
Experiment Description We first wanted to test the transaction time latency mechanism, on a few workloads, with no interaction with the XEN hypervisor. We wanted to simulate the behavior of starting a guest, as described in Section 2. So, we created 128 processes, each simulating the behavior of a guest and performing the following transaction:

- They all read the value of the same path in the database; and
- They all write in different parts of the database.

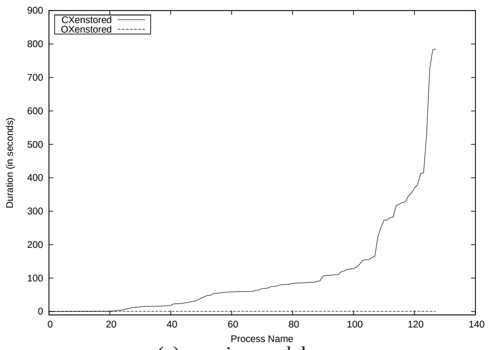
Furthermore, we repeated that transaction 500 times on each process. Then, on each process, we measured the time taken by each transaction to commit and we ordered the results.



(a) minimum delays



(b) average delays



(c) maximum delays

Figure 6. Comparison between CXenstored and OXenstored for the time taken by 128 concurrent processes to to commit 500 transactions which read from the same path and write on different ones. Processes are ordered following their completion time.

Experiment Results Part of the results of this experiment are shown in Figure 6. More precisely, these figures show for each process the time taken by:

(a) The fastest transaction to commit;

(b) The average time of the transactions to commit; and

(c) The time taken by longest transaction to commit.

Graph (a) shows that transactions are always committed faster in OXenstored than in CXenstored. Moreover, in the best case, the commit rate of OXenstored is very stable at around 700 transactions per second, as opposed to the commit rate of CXenstored which is around 300 transactions per second.

Furthermore, in the average case, Graph (b) shows that the OXenstored performance are much stable for OXenstored than for CXenstored, as the average delays for OXenstored are very low (always under 1 second), unlike CXenstored which committed few transactions over 10 seconds.

Finally, In the worst case, Graph (c) shows that CXenstored can live-lock as some commit delays are over 12 minutes. On the other hand, even in the worst case, OXenstored performance remains very stable (around 1 second).

On graphs (a) and (b), there is a small glitch around the 17th process. It is not totally clear what is the cause of that behavior.

Experiment Conclusions The obtained results clearly indicate that CXenstored cannot deal with more than 100 concurrent transactions, as opposed to OXenstored which does not show any sign of performance issues. Furthermore, when access to the database is the performance bottleneck, OXenstored is always quicker than CXenstored, even in the case of few concurrent transactions. Finally, OXenstored has a very small variance compared to CXenstored, which means that we can expect that the system will behave in a more predictable way.

7.2 Performance when interacting with the XEN hypervisor

Experiment Description Subsequently, we designed a more realistic test than the one of Section 7.1, to compare the performance of CXenstored and OXenstored to start real guests. For this experiment, we installed one guest running "Windows XP" and then we repeated the following steps 100 times:

1. clone 50 guests from the initial one (a clone is functionally equivalent to a fresh install, but it quicker);
2. sequentially start all the cloned guests;
3. sequentially shut-down all the cloned guests; and
4. uninstall (ie. destroy) all the cloned guests.

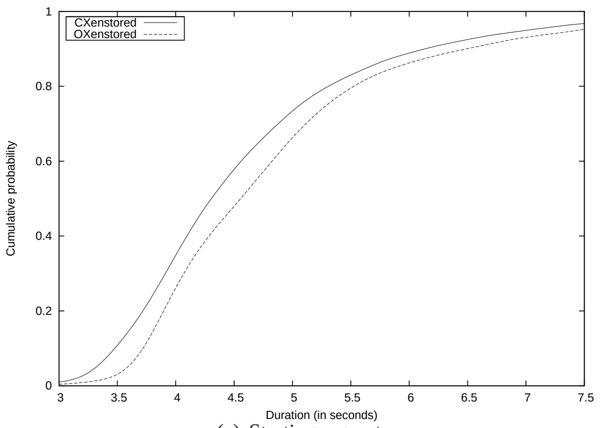
At the same time, we measured the time taken for starting and shutting-down each guest.

Experiment Results The results are shown in Figure 7. These figures show the integral of distribution probability for the time taken to complete guest start and shutdown. The results for OXenstored and CXenstored are quite similar: half the guest are started in less than 4.5 seconds and are shut down in less than 2 seconds. Furthermore, 90% of the guests are started by OXenstored and CXenstored in less than 7.5 seconds and are shut down in less than 2.9 seconds.

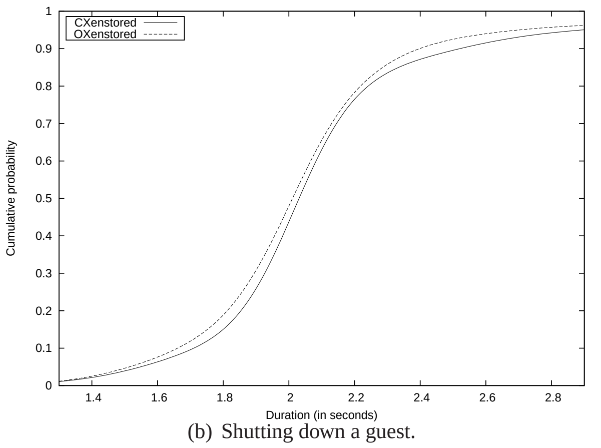
Experiment Conclusions The results we obtained show that there is no major differences between OXenstored and Xenstored in this case, that is CXenstored is clearly not a bottleneck under normal load.

7.3 Performance under load when interacting with the XEN hypervisor

Experiment Description In this experiment, we wanted to start as many guests as possible and compare the performance of CXenstored and OXenstored. Thus, we needed minimalist guests which do not use many physical resources. Hence, we used a modified version of "mini-OS", a very small operating system distributed with the XEN hypervisor sources, in order to start a lot of very small guests performing long conflicting transactions concurrently. More precisely, we created 160 "mini-OS" guests, with 1 virtual disk and



(a) Starting a guest.



(b) Shutting down a guest.

Figure 7. Integral of the distribution probability of the time taken by CXenstored and OXenstored to start a real guest.

1 virtual network each, which all do the following when they are started:

1. start a transaction;
2. write a value in `/local/domain/X/device/foo` (where `X` is the current guest ID) using the opened transaction;
3. sleep 1 second;
4. write a value in `/local/domain/X/device/bar` (where `X` is the current guest ID) using the opened transactions;
5. close the opened transaction;
6. sleep 1 second; and
7. go back to step 1.

These transactions simulate the way some monitors report statistics inside each guest (as the current memory usage for guests which are not modified to run on top of the XEN hypervisor, as “Windows” guests). We then sequentially started as many guests as we can on one host and we measured the cumulative time taken to start each of the guests.

Experiment Results Figure 8 shows the cumulative time taken to start as many guests as possible in less than 20 minutes. CXenstored begins to starts 30 guests at a constant rate of 2 seconds per guest started, but it starts to live-lock around 40 guests. Finally, after 20 minutes, it never commits the transaction which should configure and start the 70th guest. On the other hand, OXenstored keeps starting the guests at a constant rate every 2 seconds and it started all the 160 guest in less than 6 minutes.

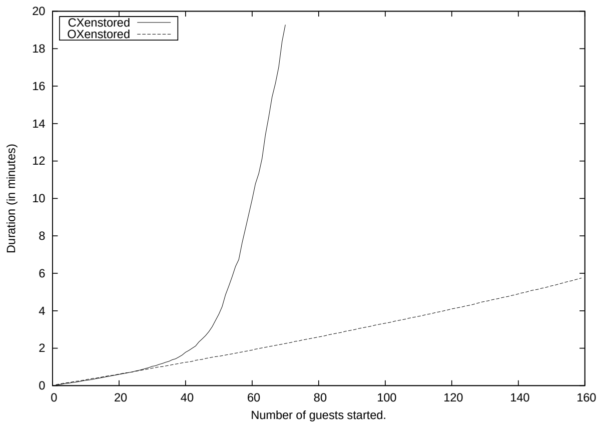


Figure 8. Comparison of the time taken by CXenstored and OXenstored to start as many mini-OS guests as possible, when these guests perform long transactions in a loop.

Experiment Conclusions Is is quite clear that OXenstored is not at all influenced by the long transactions, as opposed to CXenstored which begins to live-lock when more than 40 guests are started. Hence, the performance of OXenstored is more stable and it scales much better than CXenstored.

8. Related Work

Transactional databases have been widely studied in the last few decades (Härder and Reuter 1983; Bernstein et al. 1987). They provide to the user a very simple way of encapsulating a group of actions, called a transaction, with the following properties: If one part of the transaction fails, the entire transaction fails (atomicity); at every moment, the database remains in a consistent state: only valid data are written in the database (consistency); other operations cannot access or see the data of an intermediate state during a transaction (isolation); once the user has been notified that one of its transactions has succeeded, the transaction will persist and not be undone, even in case of a failure (durability). These properties make the concurrent programming of such systems very easy, as the user does not have to worry anymore about using locks to ensure data consistency.

The most common way to ensure atomicity in transactional databases is a mechanism called *compensation* (Gray and Reuter 1992). In case of failure, compensation consists of executing the compensating actions, corresponding to the executed actions of the failed process, in the reverse order of their execution. This approach has recently gained renewed popularity in the context of general-purpose programming known as “Software Transactional Memory” (Shavit and Touitou 1995; Harris and Fraser 2003; Ennals 2005; Riegel et al. 2006), where the purpose is to minimize the use of locks (by the use of lock-free data structures, for example). The same compensation mechanisms have also been applied successfully to build efficient and robust transactional web-services (Biswas 2004; Biswas et al. 2008). Due to the nature of the compensation mechanism (ie. keeping a list of the actions executed by each started transaction), there is a straightforward but very costly way to merge transactions: when a transaction is committed, only the started transactions that read values written by the committed transactions are aborted. Note that this may lead to further abortions of other transactions and so on, an effect called *cascading aborts*. This effect can lead to very hard-to-predict performance.

Thus, our approach based on “optimistic” transaction control, where each transaction has a local copy (called a *shadow tree* in the database literature) of the database which it is modifying, leads to much more predictable performance on tree-structured databases. Hence, a few other systems have used the same form of “optimistic” transaction control based on shadow trees, the first one of them probably being the object store of GemStone (Butterworth et al. 1991), two decades ago. However, our approach is much crisper and simpler as the applications we consider have a greater degree of locality to be exploited compared to the more general object database applications usually considered.

Furthermore, functional data-structures, including tries, have been widely studied (see the Okasaki’s book (Okasaki 1999), as example). Our axiomatization is a step forward to integrate trie properties in a theorem prover in order to demonstrate the correctness of algorithms on tries. Moreover, to the best of our knowledge our work is the first to consider the use of tries in transactional setting. Few other functional structures have been studied in that context, as functional binary search (Trinder 1989). In that work, transactions are pure functions taking an immutable database state as an argument and returning an output and a new database state. Concurrent transactions are serialized by a database manager which then uses a data dependency analysis and the referential transparency property of pure functions to efficiently schedule the transaction’s operations. Our approach is more flexible as it allows us to mix functional and imperative features when using transactions (as in *Objective Caml*).

More recently, the aim of the Harmony project (Foster et al. 2007) is to define a safe way of synchronizing different views of a shared persistent tree data-structure. The synchronization mechanism of Harmony can be seen as the commit phase of a transactional database, but where the only knowledge is the current state of the transaction, rather than a trace of modifications as in the case with compensation. The merging algorithm used by that tool needs to be specified by the users and thus can become very complex. On the other hand, our approach is much simpler and more efficient but it requires some extra knowledge about the transaction which is committed (ie. the modification prefix) and thus cannot be used in our work.

9. Conclusion

This paper describes the design of OXenstored, a successful application of functional technology in an industrial setting which demonstrates the effectiveness of a limited form of pointer comparison in a functional context. Indeed, the OXenstored database is represented and manipulated using an immutable prefix tree, for which we give a simple and formal semantics. For performance reasons, OXenstored uses reference cell comparison. This is a limited form of pointer comparison which can be elegantly integrated with the prefix-tree semantics, ensuring a safe and efficient way to merge concurrent transactions.

We also demonstrate the validity of our approach by implementing in *Objective Caml* the algorithms described in this paper and evaluating them against CXenstored, the Xenstored service written in *C* and distributed with the XEN hypervisor sources. These experimental results show that our transaction processor is not only one third of the size of of its *C* counterpart, but significantly outperforms it. As a direct consequence of these results, OXenstored will replace CXenstored in future releases of XENSERVER.

Acknowledgements

The authors would like to thank very much Jonathan Davies and Richard Sharp for making useful suggestions. They also would like to thank the anonymous ICFP reviewers whose comments greatly improved some parts of the paper.

References

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM Press, 2003.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

Debmalya Biswas. Compensation in the World of Web Services Composition. In *SWSWPC*, pages 69–80. LNCS 3387, 2004.

Debmalya Biswas, Thomas Gazagnaire, and Blaise Genest. Small logs for transactional services: Distinction is much more accurate than (positive) discrimination. *IEEE International Symposium on High-Assurance Systems Engineering*, pages 97–106, 2008.

Paul Butterworth, Allen Otis, and Jacob Stein. The gemstone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

Koen Claessen and David Sands. Observable Sharing for Functional Circuit Description. In *In Asian Computing Science Conference*, pages 62–73. Springer-Verlag, 1999.

Robert J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

Robert Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, 2005.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, 2003.

Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. `http://caml.inria.fr/ocaml/`, 1996.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

Andrew Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What’s new. In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp.*, pages 122–141. Springer-Verlag, 1993.

Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot Isolation for Software Transactional Memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

Manuel Serrano. Bee: an integrated development environment for the Scheme programming language. *Journal of Functional Programming*, 10(4):353–395, 2000.

Nir Shavit and Dan Touitou. Software Transactional Memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

Phil Trinder. *A functional database*. PhD thesis, 1989.