Strict Serializability is Harmless: A New Architecture for Enterprise Applications

Sérgio Miguel Fernandes João Cachopo
INESC-ID Lisboa / Instituto Superior Técnico, Technical University of Lisbon
{Sergio.Fernandes,Joao.Cachopo}@ist.utl.pt

Abstract

Despite many evolutions in the software architecture of enterprise applications, one thing has remained the same over the years: They still use a relational database both for data persistence and transactional support. We argue that such design, once justified by hardware limitations, endured mostly for legacy reasons and is no longer adequate for a significant portion of modern applications running in a new generation of multicore machines with very large memories.

We propose a new architecture for enterprise applications that uses a Software Transactional Memory for transactional support at the application server tier, thereby shifting the responsibility of transaction control from the database to the application server. The database tier remains in our architecture with the sole purpose of ensuring the durability of data.

With this change we are able to provide strictly serializable transaction semantics, and we do so with increased performance. Thus, we claim that strict serializability is harmless, in the sense that programmers no longer have to trade correctness for performance for a significant class of applications. We have been using this new architecture since 2005 in the development of real-world complex object-oriented applications, and we present statistical data collected over several years about the workload patterns of these applications, revealing a surprisingly low rate of write transactions.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures

General Terms Design, Performance

Keywords Enterprise Application Architecture, Software Transactional Memory, Strict Serializability, Persistence, Transactions, Object-Oriented Programming, Rich-Domain Applications, Fénix Framework

1. Introduction

The adoption of multicore architectures as the only way to increase the computational power of new generations of hardware spurred intense research on *Software Transactional Memory* (STM), leading to many advances in this area over the last eight years (for an actual and comprehensive description of the research made on STMs, see [19]).

STMs bring into the realm of programming languages, the age-old notion of transactions, well known in the area of *Database Management Systems* (DBMSs). STMs, however, are not concerned with the *durability* property of ACID, and, because of this, most, if not all, of the STMs' design decisions and implementations have little in common with their counterparts of the database world [14]: Unlike DBMSs, modern STMs are designed to scale well to machines with many cores, to provide strong correctness guarantees such as opacity [18], and to ensure nonblocking progress conditions such as lock freedom [15].

And yet, surprisingly, all of these great advances in the area of STMs have been mostly ignored by one of the communities that could benefit the most from them: The community of enterprise application development. In this paper, we show that there is much to gain by merging these two areas, namely, both in terms of development effort and of performance.

We propose a new architecture for enterprise applications that uses an STM, rather than a DBMS, to ensure the transactional semantics for its business operations. Because STMs do not guarantee the durability of the data, we show how to extend the STM to collaborate with a persistent storage system to ensure that the application's data are safely stored.

We claim, and provide evidence, that this new architecture is not only much better to develop many of today's complex, rich enterprise applications, but is also better suited to use the new generation of machines with many cores and very large memories that are increasingly running the application-server tiers of those applications.

Both this architecture and its implementation were developed over the last six years alongside the development of a real-world complex application—the FénixEDU[1] web application—and have been driving the execution of that application in a demanding production environment since 2005. In 2008, the FénixEDU web application went through a major refactoring to separate the infrastructure code from the rest of the application code, resulting in the initial version of the Fénix Framework—a Java framework to develop applications that need a transactional and persistent domain model, and that implements our proposed STM-based architecture.[2]

1.1 Main contributions

The main contributions of this paper are threefold.

First, we propose an architecture that is especially suited for the development of many of today's object-oriented enterprise applications, providing not only strictly serializable semantics to the applications' business transactions, but also improved performance over traditional implementations. Serializability alone requires that transactions be ordered as if they had executed sequentially. Strict serializability is an additional requirement on top of serializability: It imposes that transactions be serialized without reversing the order of temporally nonoverlapping transactions [24]. Intuitively, programmers can think of strictly serializable transactions as transactions that are serialized in an instant within the real-time interval in which they executed. Regarding performance, our tests show an increase of throughput in the TPC-W benchmark, for a variety of workloads and cluster configurations, up to 23.7 times, having the best results in the read-intensive workloads.

Second, we describe the key elements of an implementation of this architecture in a Java-based framework—the Fénix Framework: We describe how to extend the commit algorithm of the STM that we use to ensure the durability of the application's data; we describe the life cycle of a persistent domain object; and we describe how we use a shared, global identity map to keep domain objects in memory and preserve their identity across transactions and threads. We describe, also, the mechanisms used to allow the simultaneous existence of more than one application server accessing the same data, while still ensuring strict-serializable consistency guarantees for all the transactions running across all of the application servers.

Third, and finally, we present statistical data about the workload patterns of two real-world enterprise applications. These results were collected over an extended period of several years, with varying usage patterns, and we claim that they are an important contribution for at least two reasons. First, they provide support to the common belief (but rarely confirmed with real data) that enterprise applications have a very high read/write ratio: In both cases, the number of write transactions are, on average, only 2% of the total number of transactions, peaking below 10% for short periods of write-intensive activity. These numbers are, actually, well below the numbers that are typically observed in benchmarks such as TPC-W. Second, these results confirm our claim that strict serializability is harmless for (at least some) enterprise applications: The rate of conflicts among write transactions is almost negligible for these applications, averaging less than 0.2% of the total number of write transactions (meaning that they represent less than 0.004% of the total number of transactions).

The remainder of this paper is organized as follows. In Section 2 we argue why a new architecture for enterprise applications is needed: We believe that historical reasons have led to some limitations in the programming model, namely with regard to transactional semantics. We discuss those limitations and how they affect programmers and software development. Then, in Section 3 we describe the STM-based architecture that we propose and discuss some of its implementation details. Next, in Section 4, we describe a real-world case of a large application, as well as a smaller-sized application, both developed with this architecture. Additionally, we provide a performance comparison between two implementations of the TPC-W benchmark, one using a traditional approach and another using our architecture. We discuss related work in Section 5 and conclude in Section 6.

2. Why We Need a New Architecture

Over the last 30 years, the development of enterprise applications has evolved, influenced by diverse factors such as the changes in hardware, or the changes in the users' expectations about how applications should behave. Often, these changes had a reflection in the architecture of applications. Still, one thing has remained the same for most applications: The underlying DBMS provides not only persistence but also the transactional semantics on which application developers rely for programming business transactions.

Unfortunately, this means that developers are limited to the isolation levels provided by the underlying DBMS, which often does not provide serializability—a correctness property for concurrent transactions that is crucial to ensure the integrity of the applications' data.

Historically, DBMSs weakened the isolation level of transactions as a trade-off between correctness and performance, which resulted in the generally accepted idea that serializability is incompatible with performance. This line of reasoning is clearly illustrated by the following passage from Martin Fowler's well-known book on patterns of enterprise application architecture [16]:

> To be sure of correctness you should always use the serializable isolation level. The problem is that choosing serializable really messes up the liveness of a system, so much so that you often have to reduce serializability in order to increase throughput. You have to

[1] `http://fenix-ashes.ist.utl.pt/`

[2] `http://fenix-ashes.ist.utl.pt/trac/fenix-framework`

decide what risks you want take and make your own trade-off of errors versus performance.

You don't have to use the same isolation level for all transactions, so you should look at each transaction and decide how to balance liveness versus correctness for it.

Even though this book is from 2002, we believe that it still represents faithfully the actual state of the practice in the development of most enterprise applications. One of the problems, thus, is that the use of weaker transactional semantics burdens developers with additional, nontrivial development effort. We shall return to this problem in greater detail later in this section. Before that, however, we provide a brief historical overview of the development of client/server applications and relate it to the *status quo* of complex enterprise object-oriented application development.

2.1 The Shift From a 2-tier to a 3-tier Architecture

A core component of most modern enterprise applications is a DBMS, and its use in such applications goes a long way back. The first enterprise applications were 2-tiered—that is, they had a simple client/server architecture. The clients made their requests directly to the database server, which in turn executed the requested operations and gave back the results. This architecture was adequate in a scenario where all the clients were on a trusted network and most of the computational power resided on a big server. In this architecture, transactional control was placed in the (only) obvious location: The database server. Each client request would perform within a database transaction, and the server ensured the ACID properties. Database servers were expensive and they had to cope with a growing usage demand. Eventually, different semantics for the ACID properties appeared, which relaxed some of the properties (e.g., isolation), mostly for performance reasons. As client machines became more and more powerful, more computation could be performed on the client side, which would also take part in ensuring data consistency. Business logic consistency started to get more complex than simple low-level consistency checking (e.g., referential integrity), including complex high-level consistency (e.g., a list can only contain odd numbers). Often, in this architecture the business logic code was intertwined with the user interface code.

With the growth of the internet and the World Wide Web, a new architecture emerged. As organizations felt the need to interconnect their systems, the 2-tiered architecture no longer served their purposes. There were several reasons for this, including the systems' security and network bandwidth. The clients (now including systems in diverse geographical locations and outside the control of the intranet) were no longer trusted. The number of clients grew. The available bandwidth was far less than it was previously available on the intranet, such that sending large volumes of data, as the result of a database query, was no longer viable. This led to a 3-tiered architecture on which the server side was decomposed in two tiers: One for the application server and another for the database server. The database server was still responsible for the data persistence and ensuring transactional access. The application server was responsible for executing the complex business logic and interfacing with the clients, which would provide the user interface. This new architecture presented two main advantages over the previous: (1) the information was kept safe within the intranet and it was only accessed directly by a trusted system; and (2) large queries could be obtained and processed on the server-side before sending the results over the internet to the clients. Notably, relaxed transactional semantics was kept in place, because database performance was still an important aspect in this architecture.

As the internet grew, another revolution was taking place in the software development area: Object-oriented programming languages became mainstream and started to be used commonly in the development of large server-side applications. Characteristics such as component modularization, ease of reuse, data encapsulation, and abstraction, helped developers control the growing complexity of their applications. The adoption of *Object-Oriented Programming* (OOP), however, created a mismatch between the persistent representation of data and their in-memory representation, known as the object-relational impedance mismatch [21]. The development of *Object-Relational Mapping* (ORM) tools was the industry's answer to handle this mismatch. The purpose of an ORM tool is to take care of the data mapping between the object-oriented model and the relational model. Whereas, in part, these tools simplified the programmers' coding efforts, they also created some difficulties of their own, such as the maintenance of O/R-mapping metadata, and the varied semantics implied by object caching, lazy loading, and support for *ad hoc* queries. Additionally, different ORMs provided different semantics. Yet, the features provided by ORM tools kept depending on transactional support that was still under the responsibility of the underlying DBMS, which, in turn, still offered different flavors for ACID semantics, but none included support for strict serializability [24]. In fact, the isolation guarantees provided by databases have, for long, been a matter of confusion and discussion [2].

The natural evolution in software and hardware has led us to the current state, in which many developers depend on a 3-tiered architecture, develop application servers using an object-oriented paradigm, and rely on a relational DBMS for persistence. There are of course some exceptions to this, most notably in emerging large distributed systems, which may use different programming models [11] or different storage mechanisms [4] with different consistency models [26, 29]. These very large-scale distributed systems fit in a class of their own and, for the time being, are not the target of our study.

We concentrate on the development of complex object-oriented applications that require transactional support and data persistence. By complex, we mean applications that have a rich domain structure with many relationships among the classes, as well as business logic with nontrivial rules (but not typically a massive volume of data to process). Moreover, we assume that these applications have many more read-only than write business transactions, typically in the rate of 10 to 1 or more.

For these applications, we identify two problems with current implementations of the typical 3-tier architecture: One is the difficulty in ensuring consistency; the other is the reduced performance of the application server in the processing of complex operations. We address each problem in the following two subsections.

2.2 Consistency Problems

If given the possibility, it seems clear that every programmer would prefer to have no less than strict-serializability semantics when programming concurrent transactional operations that manipulate the state of the domain objects in their code. Having such guarantee shields programmers from concurrency hazards, and allows them to write cleaner and simpler code.

To illustrate this point consider the following example: Imagine a multi-player game where players can concurrently move their pieces in a map from one point to another with the restriction that after each movement is performed no player can be in the immediate vicinity of another player. Now consider the starting scenario depicted in Figure 1, which shows part of the map containing two players, P1 and P2.

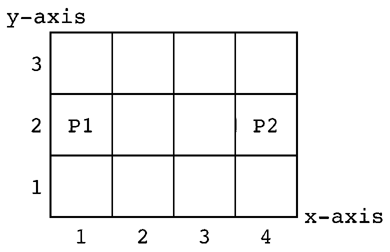

Figure 1. Two players occupying nonadjacent positions on a map.

Suppose that, concurrently, P1 will attempt to move one position to the right and P2 will attempt to move one position to the left. Only one move can succeed, because otherwise the two players would be left next to each other. A typical implementation of the `moveTo` operation could be similar to the pseudocode presented in Figure 2. The programmer checks that both the target position and its surroundings are available and, if that is the case, updates the position of the player on the map.

This code looks quite trivial and, when several moves are executed concurrently, each within its own transaction, the programmer might expect the application to work just fine—that is, after each transaction finishes, the moved player should be in a location that does not contain any adjoining players, thus maintaining the domain consistent. Sadly, this may not be the case if transactions are executed with the isolation level provided by most of today's mainstream databases, which ensure at most snapshot isolation. Under snapshot isolation a transaction may commit even if the values that were read changed in the meanwhile, as long as concurrent writes do not intersect. In this example, if transaction T1 executes `P1.move(2,2)` and transaction T2 executes `P2.move(3,2)`, then Figure 3 presents the points in the read set and write set of each transaction with regard to map coordinates.

```
class Player {
  void moveTo(int x, int y) {
    if (cell(x,y).available(this) &&
        cell(x+1,y).available(this) &&
        cell(x-1,y).available(this) &&
        cell(x,y+1).available(this) &&
        cell(x,y-1).available(this)) {
      this.currentCell().clear();
      cell(x,y).set(this);
   } else {
      throw Exception("Move not allowed");
    }
  }
}

class Cell {
  boolean available(Player p) {
    return this.isEmpty() || this.holdsPlayer(p);
  }
}
```

Figure 2. The `moveTo` operation checks the surroundings to ensure that the move is allowed and writes to the destination.

Tx	Read set	Write set
T1	(2,2), (3,2), (1,2), (2,3), (2,1)	(1,2), (2,2)
T2	(3,2), (4,2), (2,2), (3,3), (3,1)	(4,2), (3,2)

Figure 3. T1 and T2 write to adjacent places concurrently. Write sets do not intersect.

Note that write sets do not intersect, and as such, snapshot isolation will allow both transactions to commit, leading to an inconsistent domain state. This problem is well known by the name of *write skew* [12]. Yet, without changing the database concurrency control mechanisms, the current solution is to put in the programmer's shoulders the responsibility for forcefully creating a conflict between the two transactions. In this case, that may be accomplished by calling `clear()` for all of the surrounding positions, thus causing an intersection in the write sets [8]. This is definitely something undesirable from the programmer's perspective, and very much error-prone in complex applications where the potential conflicts might not be easily identified, as they may occur due to the interaction of many functionalities.

2.3 Performance Problems

So, why do not current transactional implementations change to support strict serializability? Probably, because of the generalized idea that providing strict serializability necessarily imposes unacceptable performance penalties. Such may actually be true for today's standard architectures, which ultimately rely on the DBMS for transactional support.

When ORMs were not in use or applications had simple domain models, it was common for programmers to implement a complex database query to return exactly the results sought. Generally, this meant that very few database round trips (often just one) were enough to process each unit of work requested by the client. Most of the business logic computation was embedded on the database query and handled by the DBMS. For an application with a typical 3-tier architecture, where the communication latency between the client and the server is measured in hundreds of milliseconds and the communication latency between the application server and the database takes fewer than 10ms, the time spent with the database query is almost negligible from the client's perspective.

Today, however, programmers can execute complex computations that require reading into main memory many of the application's objects. The use of ORMs and object-oriented programming facilitates and promotes object navigation instead of the creation of custom queries. This type of programming greatly increases the number of database round trips required to answer to a client's request, with negative influence on performance. If instead of one database round trip, the business transaction has to perform tens to hundreds of round trips in sequence, then the accumulated latency of all of those round trips largely exceeds the typical latency between the client and the server.

Trying to alleviate this problem, ORMs cache data on the application server tier, but still they depend on the underlying database to provide the transactional semantics. Unfortunately, developers of ORM tools have followed suit with databases in terms of the transactional properties provided to the application programmer. In fact, the use of application tier caches may further weaken the consistency semantics of the database by inadvertently providing inconsistent reads (served from the cache) within a transaction, as discussed in [25].

3. An STM-based Architecture for Enterprise Applications

The core idea in our architecture is to shift the responsibility for transactional control: It no longer lies on the persistence tier and it is shifted to the application server tier. We use an STM—more specifically the *JVSTM*[3]—to provide in-memory transactional support directly in the application server tier.

[3] `http://web.ist.utl.pt/joao.cachopo/jvstm/`

The design of the JVSTM evolved alongside the development of the architecture that we describe in this paper, and it was heavily influenced by the observation and development of real-world, domain-intensive web applications [5]. Often, these applications have rich and complex domains, both structurally and behaviorally, leading to very large transactions, but having also a very high read/write ratio (as we shall see later, but see also [5, 9] for more real-world numbers).

The new architecture that we present maintains the typical 3-tier layout, but uses the persistence tier only to ensure the *durability* property of ACID. Atomicity, consistency, and isolation are ensured at the application server tier. Conceptually, any type of storage (such as relational, column-oriented, or key-value) can be used in the persistence tier, but our initial choice was to use a relational database. We had several reasons for this decision. First, relational databases are a reliable technology for data storage. Second, they are still the mainstream backend used in many enterprise applications, as was the case of the FénixEDU web application, which had its data stored in a relational database and there were also other applications that queried the database (in read-only mode) to obtain data for several other systems, mostly for statistics. So, by using the relational model for persistence, we could migrate the FénixEDU application to the new architecture while maintaining compatibility with other legacy applications. Third, relational databases already support some level of transactional support, which enables us to simplify the commit algorithm in our solution. We have developed implementations of this architecture that run on top of other backends, namely using BerkeleyDB, HBase, and Infinispan, but in this paper we describe only the implementation that uses a relational database for the persistence tier.

The general idea is that we extend the JVSTM to make it persistent. Yet, this seemingly simple task poses several new issues. In the next subsection we provide an introduction to JVSTM and describe some of its implementation details that are relevant to the following subsections. In the remaining subsections we describe some of the most important aspects of the Fénix Framework, which implements the infrastructure that provides enterprise applications the support for performing transactional operations on shared persistent objects. We begin under the assumption that there is only one application server running. Then, in Section 3.6 we detail the changes required for this implementation to work in a clustered environment.

3.1 Brief Introduction to JVSTM

JVSTM is a Java library that implements a word-based, multi-version STM that ensures strict serializability for all of its committed transactions. Actually, JVSTM provides the even stronger correctness guarantee of *opacity* [18], which ensures that noncommitted transactions always see a consistent state of the data.

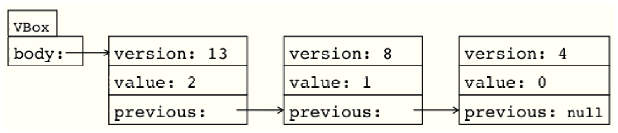

Figure 4. A transactional counter and its versions. From the programmer's perspective, the `VBox` has only one value at any given time.

JVSTM uses *Versioned Boxes* [7] to implement transactional locations in memory. A versioned box (`VBox`) holds a sequence of entries (`body`), each containing a value and a version. Each of the history's entries corresponds to a write made to the `VBox` by a successfully committed transaction. When such a writing transaction commits, it advances a global version clock and uses the new clock value to tag each of the new bodies that it creates. JVSTM's commit algorithm always keeps the sequence ordered, with the most recent committed value immediately accessible.

Figure 4 shows an example of a transactional memory location for an integer that models a counter. In the example shown, the counter has been set to zero by the transaction that committed version 4, and incremented by the transactions that committed versions 8 and 13. From the programmer's perspective there is a single memory location of the type `VBox<Integer>` that can be accessed via `get()` and `put()` methods.

A transaction reads values in a version that corresponds to the most recent version that existed when the transaction began. Thus, reads are always consistent and read-only transactions never conflict with any other, being serialized in the instant they begin as if they had atomically executed in that instant. Read-write transactions (write transactions for short), on the other hand, may conflict among them, and they require validation at commit-time to ensure that all values read during a transaction are still consistent when they try to commit—that is, values have not been changed in the meanwhile by another concurrent transaction. One of the distinctive features of the JVSTM is that read-only transactions have very low overheads and they are lock-free [15].[4]

3.2 Transactional Domain Objects

An object-oriented enterprise application represents and manipulates its persistent data as a graph of *Domain Object*s (DOs): Each DO represents an entity of the application's domain and its connections to other DOs represent relationships that it has with those DOs; we call this graph of DOs a *domain model*. In an object-oriented application, DOs are instances of some of the application's *domain classes*, and they contain a set of fields that hold the state of the object. Moreover, relationships are represented either by references to other objects or collections of objects, depending on the multiplicity of the relationship. Naturally, both the DOs' state and their relationships may change over time, as a result of the application's operations, which execute concurrently by reading and changing the domain model.

Thus, as DOs are mutable and shared, they need to be transactional. As we saw before, JVSTM provides us already with transactional memory locations—the versioned boxes—but we need to have transactional DOs instead. To create transactional DOs, we wrap all of the mutable state of the domain model with versioned boxes, which, themselves, may contain only immutable objects. So, if, for instance, we have a DO representing a person with a name and this name may change, we will use a versioned box to hold the person's name and we will keep this box within the instance of the class `Person` that corresponds to our DO.

To implement this approach correctly, all of the fields of a domain class must be `final` and their types must be either `VBox` or an immutable type (such as `String`, `Integer`, or any of the Java's primitive types). The contents of each `VBox`, on the other hand, must always be an instance of an immutable type. Note that all of the types corresponding to domain classes are immutable types also, and so we may have boxes that contain DOs.

The key result of following these rules is that DOs are now transactionally safe: DOs may be shared among all of the concurrent threads running JVSTM transactions within the application, thereby ensuring a strictly serializable semantics for all of the application's business transactions. Moreover, as we shall see in the following section, having transactional DOs will allow us to reduce both the execution time and the memory consumption of the application.

A potential problem with our approach, however, is that the implementation of domain classes that strictly adhere to the above rules is error prone, if done manually by programmers. This problem is avoided in our approach by providing a domain-specific language [17] to describe the structural aspects of a domain model—the *Domain Modeling Language* (DML) [5, 6]. DML has a programmer-friendly, Java-like syntax to describe entity types, their attributes, and relationships among them. So, a programmer using the Fénix Framework creates the application's domain model using DML, and then the DML compiler automatically generates the source code corresponding to the structural aspects of the domain classes; the behavioral code is separately developed in plain Java by the programmer.

Besides simplifying the task of implementing the domain classes, this approach based on code generation has another important benefit: It allows us to change, and eventually fine tune, the layout of DOs. For instance, in Figure 5, we show an example of two possible layouts for an hypothetical domain class `Person`. In the first layout, all of the DO's attributes are kept in a single versioned box, whereas in the second layout there is one versioned box per attribute. This versatility enables adaptive designs that take into account

[4] In fact, apart from the creation of a new transaction instance, read-only transactions run entirely wait-free [20].

```
            // using one versioned box per object
class Person {
  final VBox<Person_State> state = new VBox<Person_State>();

  private class Person_State {
    String firstName;
    String lastName;
    Address contact;
  }

  String getFirstName() {
    return this.state.get().firstName;
  }

  void setFirstName(String firstName) {
    Person_State newState = this.state.get().clone();

    newState.firstName = firstName;
    this.state.put(newState);
  }
  [...]
}


            // using one versioned box per attribute
class Person {
  final VBox<String> firstName = new VBox<String>();
  final VBox<String> lastName = new VBox<String>();
  final VBox<Address> contact = new VBox<Address>();

  String getFirstName() {
    return this.firstName.get();
  }

  void setFirstName(String firstName) {
    this.firstName.put(firstName);
  }
  [...]
}
```

Figure 5. Two possible memory layouts for the same persistent data. Also shown are the generated `get` and `set` operations for the `firstName` attribute, in both cases.

application-specific characteristics regarding how domain data are accessed. Objects that are seldom modified can have their entire state in a single versioned box, which reduces memory usage. Objects with high contention can benefit from having one versioned box per attribute, which will reduce the number of conflicts between transactions that manipulate different attributes of the same objects.

3.3 The Domain Object Cache: Domain Objects Have Identity

In complex object-oriented applications where business transactions have to traverse deep graphs of objects, the resulting database round trips typically incur into an unacceptable performance cost. Reducing the number of database round trips, becomes, thus, essential for performance. We address this problem in the Fénix Framework, by using a global *Domain Object Cache*, shared by all threads, which maintains DOs in memory for as long as possible. The key idea is that read-only transactions should access the database only when DOs are not available in memory, which should not happen often if most of the application's data fit in memory. The goal is that, once loaded, DOs will remain in memory until the Java garbage collector needs space that it cannot find in any other way. Only in that case, may DOs be removed from the cache and garbage collected (provided that they are not in use by any transaction). In our current implementation, we rely on Java's `SoftReferences` for this behavior.

One of the results of using this approach for applications whose entire persistent data fit in memory is that, after warming up, the application server will never access the database for read-only transactions.[5]

The implementation of the Domain Object Cache follows the Identity Map architectural pattern [16]. We use only one instance of the Domain Object Cache per instance of the application server. Thus, the Domain Object Cache is shared among all threads (and, consequently, among all transactions) that execute in the application server. Notice that this domain object caching behavior is very different from that provided by most, if not all, of the current ORM implementations. All ORMs that we know of ensure transaction isolation by delivering different copies of the same object to different transactions. In such implementations, new objects are constantly allocated and deallocated, as more transactions execute. This happens, regardless of whether the object came from the database anew or was read from some second-level in-memory cache. The Domain Object Cache of our architecture ensures that, at any given time, the same persistent object is represented by at most one (reachable) instance in memory. As we saw before, DOs can be shared among all transactions because they are transactionally safe. Thus, unlike what happens with ORMs, in our approach the memory required to hold DOs does not depend on the number of concurrently running transactions.

Each entry in the Domain Object Cache maps an object's unique identifier (*OID*) to a reference to the object itself. The OID is assigned automatically by the framework, ensuring its uniqueness, when a domain object is created and remains unchanged thereafter. The reference to the object is a Java `SoftReference`, which allows for cached objects that are no longer referenced elsewhere to be garbage collected when their memory is necessary, but will keep objects in memory while they fit.

The *lookup* and *cache* operations are performed automatically by the infrastructure's implementation during an object's lifecycle. When a new object is allocated it is automatically cached before being made available to the application. This is so, regardless of whether the object is a new instance or an already existing object being materialized from persistence. Next, we detail the object allocation mechanism.

3.4 Domain Object Allocation and Loading

An application's DO is allocated either when the program creates a new instance, or when an existing DO that is not in main memory is requested by its OID.

[5] There is an exception to this rule when multiple application servers share the same database. This is discussed further ahead in Section 3.6.

In the first case we simply cache the new instance before returning it to the application, as we already mentioned. Even if the transaction in which the DO was created aborts, there is no problem in having cached it, because OIDs are unique and therefore the DO may linger in cache for a while until it is eventually garbage collected: Its cache entry will never conflict with another.

In the second case, when an existing DO is requested by its OID, we perform a cache lookup. If the DO is found, then it is simply returned. Otherwise, we allocate the DO, put it in the cache, and return it to the application.

In Java, object allocation is tightly coupled with object instantiation. To implement our allocation mechanism we perform bytecode rewriting in all DO classes to inject a special constructor that is used exclusively by the framework whenever an object needs to be materialized in memory.[6]

This allocation mechanism completely avoids database access. Such is possible because the OID encodes information about the DO's concrete class, which enables us to allocate a DO of the correct class without knowing the value of its fields yet. Only if the application later requests the value of any of the DO's fields, will a load from the database be triggered. This corresponds to the *Lazy Load pattern using ghosts* [16]. Each ghost is not a proxy: It is the object itself. The difference is that its state is not yet known, because all of its versioned boxes are created and set to a special empty value (a `NOT_LOADED` flag). When any value of its attributes is actually accessed, the transactional system identifies the special flag and triggers a database load of the missing value. So, using ghosts maintains the Domain Object Cache's property that, at any given time, the same persistent object is represented by at most one (reachable) instance in memory.

The difficulty with lazy loading the contents of a DO lies in the fact that the in-memory representation of the attributes is versioned (recall that all mutable state is kept in versioned boxes), whereas the persistent representation in the relational database stores only the most recent committed state for any datum.[7] So, we need to ensure than when a running transaction, for example T1, lazily reads a value from the database, it will see a consistent value, and not some other value that another committed transaction may have written after T1 had started. We solve this by (1) opening a database transaction in the beginning of the application server's memory transaction, (2) keeping it open for the duration of the memory transaction, and (3) requiring that the underlying database supports *snapshot isolation* [2], which is common nowadays. With this we ensure that when the application needs to load additional information from persistence it will always see a consistent snapshot corresponding to the most recent version that existed when the transaction started. The most straightforward implementation of this strategy requires a database transaction for every memory transaction. This need can be greatly reduced in two ways: One is to share database transactions among all memory transactions that start in the same version. This is possible because the database transaction is only used to read data. Writes are performed in a separate database transaction during the commit of the memory transaction (cf. Section 3.5). The other way is to not always start a database transaction in the beginning of the memory transaction. If it is likely that all the required information is already loaded in memory, then database access is really not needed. In the event that a memory transaction needs something from the database and no database transaction has been open yet, it is enough to restart the memory transaction, and this time to open a database transaction in the beginning: Using this strategy can pay off when most of the database fits into main memory, and the probability of a missed datum is low. We are currently not using any of these techniques, and we chose to open a database transaction for every memory transaction.

Generically, this mechanism allows us to materialize in memory any object given its OID without having to load it from the database. A typical use case in an object-oriented application is to load some *root* object and then navigate through the object graph by reading the objects' attributes. This way, objects are allocated in memory as soon as they are referenced, but their contents are only loaded from persistence when they are accessed for the first time. Moreover, if different paths in the object graph lead to the same object, they will actually lead to the same instance, not to another copy. This is true regardless of the thread from which the object is accessed.

3.5 Persisting Domain Objects

The JVSTM deals with all the in-memory transactional support: It provides atomicity and isolation with strict-serializability semantics, but it does not provide any form of data persistence. It follows a *redo log* model for STMs [19], which means that it keeps the write set in a transaction-local context, and it applies changes to the corresponding shared transactional locations only during the commit operation if the transaction is valid to commit. A transaction T is valid to commit if its read set does not intersect with the write set of any other transaction that committed meanwhile (between the instant T starts and the instant T commits).

Therefore we need to ensure that the commit of a valid transaction is both persisted and made visible in shared memory, atomically. The transactional API provided to the programmer overrides the `commit` operation from the JVSTM to add the required behavior.

Figure 6 presents the commit algorithm for write transactions (the commit of read-only transactions simply returns). Committing a transaction is composed of three stages: Read

[6] The main reason to inject this constructor, instead of generating it in the source code, is to ensure that it will not include any instance initialization code that the programmer may have added to the class.

[7] Of course that this mismatch could easily be solved by storing all the versions in the database, but that would go against the previous design decision to use the typical relational data structure, and would consequently break compatibility with legacy applications that queried the database.

```
1  class Transaction {
2    void commit() {
3      GLOBAL_LOCK.lock();
4      try {
5        if (validate()) { // throws exception if validation fails
6          int newVersion = globalClock + 1;
7          persistChanges();
8          writeBackToMemory(newVersion);
9          globalClock = newVersion;
10       }
11     } finally {
12       GLOBAL_LOCK.unlock();
13     }
14   }
15
16   void persistChanges() {
17     // The database transaction provides write atomicity
18     beginNewDBTx();
19     writeBackToDB();
20     commitDBTx();
21   }
22 }
```

Figure 6. We extend the `commit` operation of the JVSTM to write the changes to persistence, after validating the transaction in memory.

set validation, writing back changes to shared locations, and publishing changes globally. The global lock provides mutual exclusion among all committing write transactions, which trivially ensures atomicity between validating the read set (line 5) and writing back the write set (line 8). Also, the version number is provided by a unique global version clock that only changes inside the critical region. In the commit algorithm of the JVSTM the linearization point occurs when the global clock is updated (line 9). In this new version of the commit algorithm, however, the effects of the commit are visible as soon as the `persistChanges` finishes. After that point, the changes made by the transaction are seen by other transactions that start henceforth, because they open a connection to the database and read the most recent state.

The shaded code in line 7 extends the JVSTM's commit with an additional step that sends changes to the persistence tier. The order of invocation of the `persistChanges` operation is critical to ensure the required transactional behavior: It is performed after the transaction has been validated by the STM, and before actually making the changes visible in memory. The former ensures that the state to persist is consistent, and the latter ensures that a possible failure in the `persistChanges` operation occurs before modifying any shared memory. Because writing to persistence is performed within the global commit lock, a database transaction never conflicts. If the database write fails for some catastrophic reason (such as loosing connection to the database), then the transaction is aborted in memory. We do not depend on the database transactional semantics for any transaction to succeed. However, it is necessary that the `persistChanges` operation performs an atomic write to the database, to account for any catastrophic crash of the application server during the write to persistence. If this happens, then when the application server restarts it will see a consistent state, either before or after the current commit operation. Again, after persisting changes, the remainder of the commit algorithm can only fail for catastrophic reasons (such as running out of memory), in which case the application server's restart will certainly see the current transaction as committed in the persistence.

When committing a memory transaction, a database transaction is already undergoing, because, as we mentioned in Section 3.4, such is used to ensure consistent reads from the database. For that reason, the `beginNewDBTx` operation will relinquish any database transaction that may be in use and will create a new transaction to write the changes. Therefore the only database transactions that update the contents of the database are always created inside the global commit lock.

3.6 Clustered Environment

Up until this point we have presented the architecture under the assumption that there is only one instance of the application server running. Multicore computers with large amounts of available memory are becoming mainstream and they are the natural target hardware for applications developed with the architecture that we propose: (1) The typical workloads are highly parallel, which benefit from the increasing number of cores, and (2) having lots of memory enables the program to keep most, if not all, of its data in memory, thus reducing the number of database roundtrips. However, even when a single computer built with this commodity hardware is enough to run the entire application server with good performance, there are other reasons for deploying in a clustered application server environment, for example to enable fault-tolerance, or even to enable live hardware upgrades. In this subsection we present the additional changes to the current infrastructure to support more than one application server.

To support multiple application servers we need a mechanism that enables the servers to synchronize their changes, namely updates to the global clock and to the application's shared state. We decided to use the persistence tier to communicate the state synchronization information between the application servers. This was a relatively straightforward decision given that the communication mechanisms between the application and persistence tiers already existed.

As we have previously shown, state changes occur only during the commit of write transactions. The commits are already ordered within one application server because of the global (per application server) commit lock. Additionally, the `commit` operation needs to provide ordering between two concurrent commits from different servers, and to write a *change log* that can be used by the other servers to update their state. Each change log associates the commit version with the identification of the versioned boxes that have changed in that version. The change log is similar to a write set, except that it does not actually contain the values written, only the identification of the versioned boxes where they were written, along with the version number. Figure 7 shows

```
1  class Transaction {
2    void persistChanges() {
3      beginNewDBTx();
4      boolean dbCommit = false;
5      try {
6        if (updateFromChangeLogs(true)) {
7          validate();
8        }
9        writeBackToDB();
10       writeChangeLog();
11       commitDBTx();
12       dbCommit = true;
13     } finally {
14       if ( ! dbCommit) {
15         abortDBTx();
16         throw CommitException;
17       }
18     }
19   }
20
21   boolean updateFromChangeLogs(boolean needLock) {
22     if (needLock) {
23       SELECT ... FOR UPDATE
24     } else {
25       SELECT ...
26     }
27     // process result-set and return whether state changed
28  }
29 }
```

Figure 7. The `persistChanges` operation ensures cluster-wide serialization by relying on a database lock and validating changes against the most recent committed state.

the `persistChanges` operation with the required modifications.

After beginning a new database transaction the application server starts by requesting an update to its state (line 6). The `updateFromChangeLogs` queries the infrastructural table that holds the change logs. The `needLock` flag set to `true` causes the execution of a *SELECT* ... *FOR UPDATE* SQL query. Given that every committing transaction performs the same database request, this query effectively obtains a cluster-wide lock, and establishes the necessary total order for all commits. The lock is held until the database transaction finishes. The result of the query is the list of change logs. Next, these changes are applied in memory: For each committed transaction, a new version is added to the corresponding versioned boxes, setting the new value with the special `NOT_LOADED` flag. By doing so, rather than loading the most recent value of every versioned box that has changed, we avoid additional database queries within the critical region. In fact, these values may never be needed in this application server, but if they are, when a transaction needs them, the normal database load will be triggered, as discussed in Section 3.4. Finally, the `updateFromChangeLogs` returns an indication of whether it performed any state changes. If so, then the transactional system now needs to revalidate the memory transaction, because the previous validation (Figure 6, line 5) was performed in an earlier state. For code correction, it would be enough to perform validation only once, after checking the database for any updates. However, database access is much more expensive than in-memory validation, so the first validation allows for quick early conflict detection and, thus, can avoid the cost of a database access for a transaction that is found to be already invalid with the existing in-memory state.

Next, the commit proceeds as before, writing back the changes of the current transaction. Additionally, it writes the change log for the current transaction. When the database transaction commits, changes will be visible to other application servers. As before, if something fails (noncatastrophically) during the `commit` and before the database transaction commits, the system will abort the transaction. After the database commit, the only possible causes for failure are considered catastrophic for the application server, and when it restarts it will view the transaction as committed.

Even though state changes only occur during a commit, their occurrence must be taken into account also during the start of a new transaction. In a clustered environment, every new transaction must check the change log for possible updates from other servers. This is required to ensure that future database reads are consistent with the global version clock that the application server keeps. Recall that we do not store versions in the database. So, when a new memory transaction begins and opens a database transaction (to ensure consistent reads during loads from the database), it will see the most recent committed data. Thus, it needs to update the state in memory from the change log; otherwise it would load data with the wrong version.

There is yet another important reason to update from the change log in the beginning of every transaction: To ensure strict serializability for the clustered system. As an example, suppose that in a cluster of two application servers, a client application performs an **update** request (R1) to an application server (AS1), which is **observed** to have been completed **before** another client application executes another request (R2) to the other application server (AS2). We can ensure that the execution of R2 will already see the effects of R1, because the update from the change log in the beginning of the second transaction will necessarily occur after the database commit of the first transaction. Figure 8 presents the regular `begin` operation extended to invoke `updateFromChangeLogs`. The difference in this invocation from the one performed inside the `commit` is that it sets the `needLock` flag to `false`: It does not need to acquire the cluster-wide lock, because it will only read from the change log. However, because reading the change log may lead to updates in DOs, these updates need to acquire the same lock as the commit (Figure 6, line 3).

4. Evaluation

Since its extraction from the FénixEDU web application, the Fénix Framework has been used to develop several other real-world applications, but FénixEDU is still the largest application that we know of that uses the approach that we propose in this paper.

```
class Transaction {
  void begin() {
    beginNewDBTx();
    updateFromChangeLogs(false);
    // rest of normal begin
  }
}
```

Figure 8. The beginning of a transaction in a clustered environment, not only opens a database transaction to ensure snapshot isolation for loading, but it also updates the application server's state with the most recent commits.

In the following subsection we present some of the statistical data collected over the last years both for FénixEDU and for •IST (read as "dot IST"), which is another real-world web application that started to be developed with the Fénix Framework in 2008. These statistical data allow us to analyze the typical workload of these applications, which, as we shall see, have very high read/write ratios and a remarkably low rate of conflicts. The goals of this initial evaluation subsection are twofold. First, it allows us to show that it is feasible to ensure strict serializability for all business transactions without incurring into many conflicts, which could lead to poor performance of the system. Second, it provides us with extensive data regarding workload patterns for two, independently developed, real-world complex web applications, adding real evidence to the general belief that such applications have many more reads than writes.

To evaluate the performance benefits of our architecture, however, these real-world applications are not the best fit. Even though performance problems in the FénixEDU application were one of the primary reasons for developing this architecture, we do not have quantitative measurements of the performance benefits of the new architecture for FénixEDU: We have only anecdotal evidence from its users that the performance of the application increased significantly once the new architecture was adopted, even though the load of the system increased steadily overtime and the hardware remained the same.

Unfortunately, as in most real-world scenarios, it is overly complex and expensive to reimplement these applications with a traditional architecture, so that we could compare the two alternatives. Instead, we chose a standard JDBC-based implementation of a widely known application server benchmark, the TPC-W, and reimplemented it using the Fénix Framework. In subsection 4.2 we use the two implementations of this benchmark to do a more thorough performance evaluation of our architecture.

4.1 Workload Patterns of Real-world Applications using the Fénix Framework

FénixEDU is a large web application deployed as part of an academic information system for Higher Education developed at *Instituto Superior Técnico* (IST), the largest school of engineering in Lisbon, Portugal. IST is home to more than 6,000 undergraduate students (BSc), 4,000 graduate students (MSc and PhD), and around 1,100 faculty members and researchers. FénixEDU supports the majority of IST's web-based functionalities for the entire school ranging from courses and academic management to administrative support, scientific support, and admissions. The functionalities it provides can be as simple as logging a summary for a class or as complex as generating and validating timetables for the entire school.

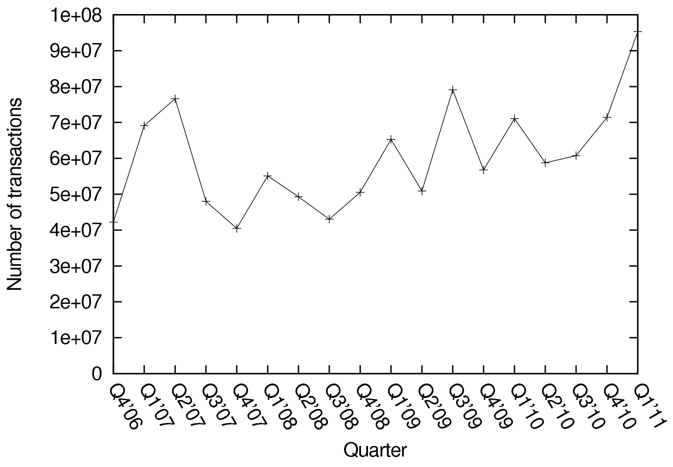

Figure 9. Total number of transactions per quarter in FénixEDU.

The development of FénixEDU begun in 2002, following the at-the-time best practices of software development and engineering: It was based on a traditional web application architecture. Following a rapid evolution of its feature set, with an ever increasing number of users, in late 2003 the application started to have not only performance problems, but was also facing development problems due to the complexity of the programing model, which was compounded by the pressure put on developers to make the application perform better. These problems urged an architectural shift to the architecture that we described in this paper.

Since September 2005, the application has been running with this new architecture, which has since then been extracted into the Fénix Framework. Currently, the FénixEDU web application contains approximately 1.2 million lines of code, over 8,000 classes, of which more than 1,200 represent domain entities described in the DML. It has over 3,600 different web pages for user interaction. Every 5 minutes, FénixEDU logs statistics about its operation. It is by far the largest application using the Fénix Framework, and the one from which we have collected the most statistical data.

Figure 9 shows the evolution in the total number of transactions processed per quarter since the last quarter of 2006. Overall, the number of transactions has been increasing, which we can relate to the continuous increase in the functionalities provided by the system to its users. The fluctuations occur mostly because users' activity is not constant

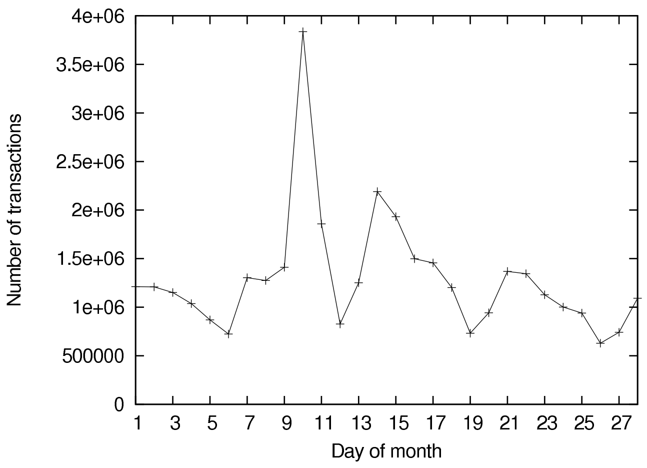

Figure 10. Daily number of transactions in FénixEDU during February 2011.

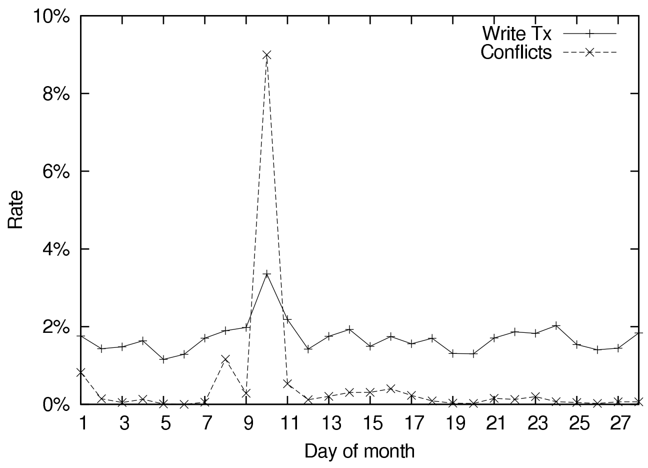

Figure 11. Daily rate of writes and conflicts in FénixEDU during February 2011.

over the year. For example, during vacation periods, activity drops considerably.

FénixEDU processes a daily average of 1 million transactions during work days. The peak usage of the system occurs twice a year, when the enrollment period for the next semester opens, at which time nearly 10,000 students hit the system at the same time. Figure 10 shows one such peak during last February. During the entire day of February 10, 2011, the system processed 3.7 million transactions. However, enrollments only started at 6:00 P.M. and in the following 60 minutes the system processed 1.1 million transactions, which amounts to a peak of more than 300 transactions per second.

In Figure 11 we present the daily rate of write transactions and conflicts, also for February 2011. Notice that under normal load, over 98% of the total number of transactions processed by the system are read-only transactions, and of the remaining 2% (the write transactions) there are on average less than 0.2% restarts due to a conflict. At peak times, the rate of write transactions goes up, but still remains under 4% and the conflicts rise to about 9% (of the write transactions).

Note, however, that this throughput is not limited by the hardware, but merely reflects the demand made to the system by its users. In fact, all this is run on a cluster of two machines (for fault-tolerance) equipped with 2 quad-core CPUs and 32GB of RAM each that are under-used. Data loaded in memory usually take approximately 6GB, whereas the relational database size (measured by MySQL) is under 20GB. This shows that it is possible to deploy a real-world application running under strict-serializability semantics without a negative effect on performance. We believe that these characteristics of the FénixEDU web application are not uncommon, and that, in fact, are representative of a large fraction of modern enterprise applications, for which our new architecture provides a very good fit.

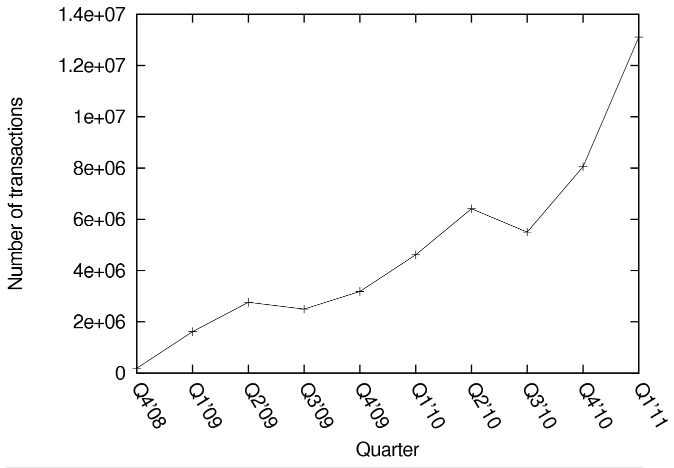

Figure 12. Total number of transactions per quarter in •IST.

More recently, in mid 2008, IST begun the development of another web application, named •IST. The goal of this new web application is to support many of IST's workflow processes. It includes the management of several administrative tasks, such as acquisition processes, travel authorizations, administrative staff evaluation, internal staff transfers, and document management, among others. It is dedicated to support the work of faculty members, researchers, and administrative staff. Because it does not include the students, its user base is much smaller than FénixEDU's, but is used in a more regular fashion by its users.

In Figure 13 we show the evolution in the total number of transactions processed per quarter, since the initial deployment of this application. As we collected statistical data for the entire lifetime of this application, we may see a steady increase of the application's usage, since its early beta stages in the last quarter of 2008, when both few processes were supported and few users were using the application. Since then, the application has been opened to its entire user base

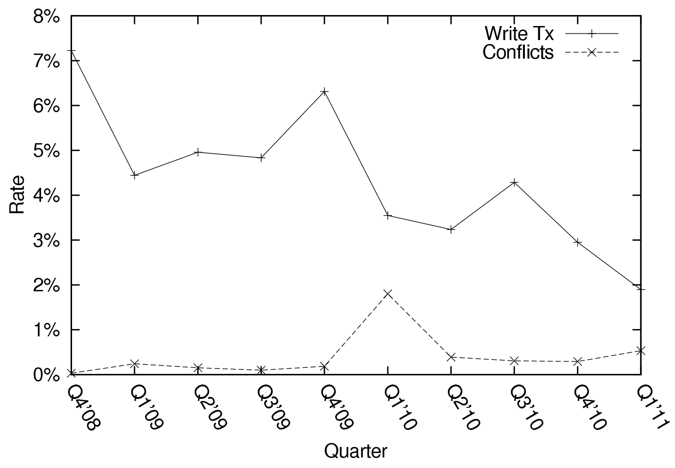

Figure 13. Quarterly rate of writes and conflicts in •IST.

and has also included many more features, leading to a significantly higher number of transactions processed in recent quarters. It is expected that this growth may continue for a while as new features are added. However, it may eventually reach a plateau, because of the limited number of users.

Unlike the FénixEDU, the •IST does not have any publicly accessible web pages, other than the login page. The functionalities it provides tend to increase the number of write transactions, because many of the operations provided to the users involve the execution of workflow steps that cause changes to the application state. Figure 13 presents the rate of write transactions and conflicts in this application, and provides some insightful information. First, we confirm that, as expected, the percentage of write transactions is higher than in FénixEDU. However, it shows a tendency towards decreasing. In fact, the absolute number of write transactions per quarter (not shown in the plots) has been increasing, but so has the number of read-only transactions. And the latter, have increased much more. We believe that this is a natural consequence of having more data available in the system, because more users access the system to check the status of the workflows in which they take part. So, in fact we draw a very interesting conclusion from this observation, and it reinforces our belief that, in this kind of applications, the reads largely outnumber the writes. Even when a web application is more geared towards operations that involve having its users making changes to the application's data, as in •IST, their users tend to execute many more of the read operations. Finally, despite the higher rate of writes, the conflicts remain close to zero. The abnormal spike in the first quarter of 2010 is due to the execution of large migration scripts that executed scattered throughout the weeks and often conflicted with users's activities. Nevertheless, it represents a very low percentage of the write transaction.

4.2 Performance Comparison with a Standard JDBC Architecture

The TPC-W is a benchmark created by the *Transaction Processing Performance Council* (TPC) [8]. It implements a web commerce application for an online bookstore. It has the typical 3-tiered architecture, where the clients (a set of emulated web browsers) access a web server to browse and buy books. The state of the application is stored persistently in a relational database. The TPC-W has already been discontinued by the TPC, but nevertheless it still provides a simulation of the class of applications that we are interested in. Moreover, there was already an open-source implementation available[9] in Java that we could immediately use.

We started from the JDBC-based implementation and modified it to implement our architecture. The detailed changes are documented online [10], together with instructions for running both implementations. In short, we defined the object-oriented domain model of the application using the DML, and then implemented the functionalities that corresponded to the SQL queries found in the original version. The browser emulator for the client remained unchanged, as we completely reused the presentation layer of the application server (developed using Java Servlets).

The primary performance metric of the TPC-W measures throughput as the number of *Web Interactions Per Second* (WIPS). Each successful request/response interaction between the client and the server counts as one web interaction, so the higher the WIPS the better the performance.

We compared the average WIPS obtained by the two implementations of TPC-W (FF-based and JDBC-based) in different scenarios. We used two interaction mixes from the TPC-W specification: The *browsing mix*, and the *shopping mix*. The former mix performs browsing (read-only) operations approximately 95% of the time, whereas the latter performs browsing only for 80% of the time. The rest of the time the clients perform shopping requests that execute write transactions. Besides the mixes specified by the TPC, we created a mix of our own, which consisted of 100% browsing interactions (*read-only mix*). Despite not representing the common case, this scenario is relevant to us, because it exercises the best possible throughput for our architecture, given that it minimizes the required database round trips.

Our architecture was designed to work best on a single application server with many cores and large memory heaps. So, the first group of tests were performed running a single application server in a machine with a NUMA architecture, built from four AMD Opteron 6168 processors. Each processor contains 12 cores, thus totaling 48 cores. The system had 128GB of RAM, more than enough to keep data from all processes (clients, application server and database) loaded

[8] `http://www.tpc.org`

[9] `http://tpcw.deadpixel.de/`

[10] `http://www.esw.inesc-id.pt/permalinks/fenix-framework-tpcw`

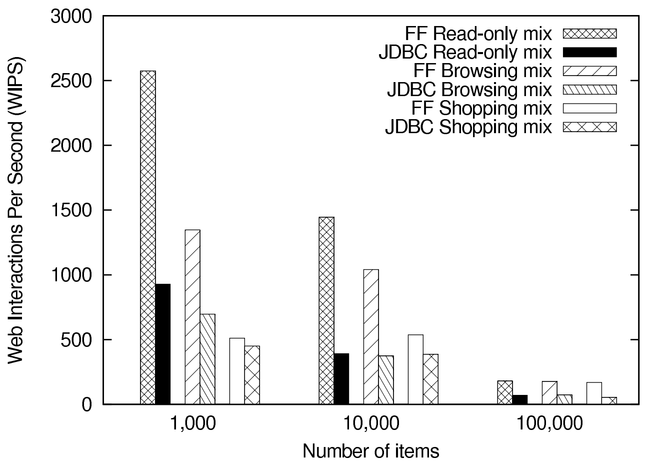

Figure 14. Throughput for a single application server, with 10 concurrent clients, varying the number of items in the database. Our implementation consistently shows a higher throughput.

in main memory. The largest amount of memory ever required by the application server, for a single test case, never exceeded 15GB. We tested using Java 6 with HotSpot(TM) 64-Bit Server VM (19.1-b02). The database was MySQL 5.1 and the application server was installed as a web application on Apache Tomcat 6.

For the first test the database was populated with three different data sets of 1,000, 10,000, and 100,000 books, respectively. The WIPS we obtained for all the configurations tested are shown in Figure 14. These are the average WIPS achieved when running with 10 clients that concurrently perform requests to the application server. Each client performed its own requests sequentially, with each request being performed immediately after receiving the server's response (client's configuration parameter `THINK_TIME` $= 0$). This value does not simulate the interaction patterns of real users, which always spend some time between consecutive interactions, but our goal was to produce an uninterrupted flow of requests to stress test the application server, and measure how many WIPS it could process under the given load.

The implementation based on the Fénix Framework shows a higher throughput than the JDBC-based implementation for all tests. It shows on average 2.4 times more WIPS, and up to 3.7 times more in the case of the read-only workload for 10,000 items. These results clearly show that it is possible to provide the programmer with strict-serializability semantics and an increased application performance.

Despite these good results, we believe that they could be even better, because in our version of the TPC-W with the Fénix Framework, we tried to do the most straightforward implementation that closely mimicked the original domain structure and SQL queries. In doing so, we have often produced an implementation that is not optimized. As an example, consider the operation that lists the best sellers. This operation requires looking at the 3333 most recent orders. To speed up this web interaction we could have easily maintained a list with the most recent orders, but we did not. Instead, we iterated through the list of all orders every time to produce the list of most recent orders. Note that the original SQL query (`...WHERE orders.o_id` $>$ `(SELECT MAX(o_id)-3333 FROM orders)...`) performs much quicker, because the `o_id` column is the primary key, thus indexed. This kind of optimization is tempting for at least two reasons. First, it is trivial to implement. Second, it is something that a programmer of an object-oriented application would probably do, given the requirement to compute the best sellers. Arguably, we could have made an implementation that would be even faster than it already is, if we had built the application's domain model from scratch, rather than adapted the SQL-based implementation.

Mix	WIPS	
	FF	JDBC
Read-only	1904.00	159.68
Browsing	659.77	151.71
Shopping	341.58	279.29

Table 1. Absolute WIPS for 10 clients. The baseline values for our implementation already start atop of the values for the original implementation.

We identify two reasons for our current performance gains. One reason has to do with the reduction in database queries: Once loaded from the database, data remain in the application server's memory for as long as possible, whereas in the original implementation every client request requires at least one database query to get the information needed. The other reason is that, in our transactional system, read-only transactions run completely unaffected by other transactions.[11] Being the majority of operations read-only and given that the entire database fits into main memory, most of the requests can be answered much faster in our implementation.

In the next set of tests we intended to measure how the two implementations scale for an increasing number of clients. We populated the database with 1,000 books and 172,800 clients,[12] and tested the application with the number of concurrent clients ranging from 10 to 60 in increments of 10 clients per test. Recall that the hardware provides 48 cores. The work performed by the clients is almost negligible. Most of the computation time is spent either in the application server or in the database processes.

Figure 15 shows the speed-up obtained for 20 to 60 clients. We calculated the speed-up taking as the baseline the WIPS obtained with 10 clients for each test scenario, which are shown in Table 1. Notice that the absolute WIPS

[11] Write transactions contend for the global commit lock, during the commit phase only.

[12] According to the TPC-W specification, the database must be populated with 2880 $\times$ c clients in order to use up to c concurrent clients.

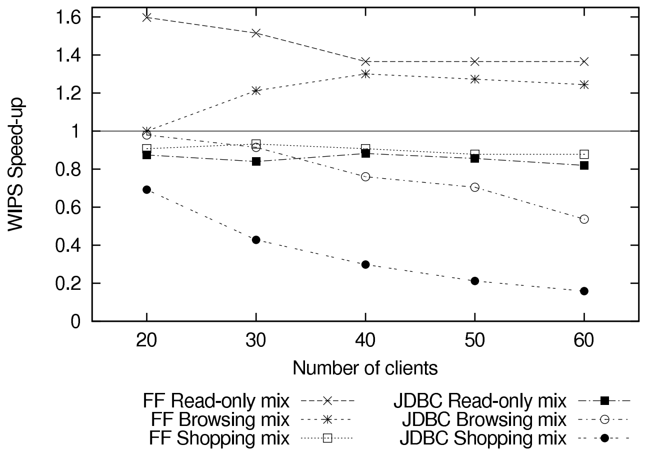

Figure 15. Speed-up for a single application server and a varying number of concurrent clients. The baseline is the throughput for 10 clients.

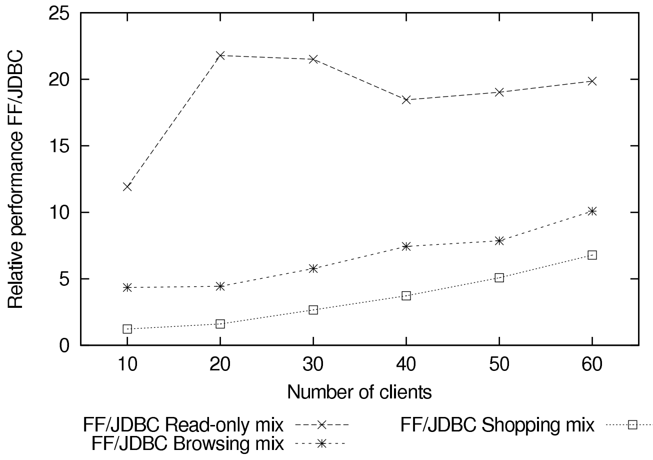

Figure 16. Performance of the FF-based implementation relative to the JDBC-based version.

shown in this table are fewer than those shown in Figure 14 for 1,000 books, because for this test the database had to be populated to accommodate up to 60 concurrent clients, which increased the database size and, consequently, the time per request, thus lowering throughput.

From Figure 15 it is clear that our solution makes better use of the available hardware parallelism: For the read-only and shopping mixes, throughput increases when compared to the baseline. It stabilizes around 1.4 even tough the read-only mix is capable of up to 1.6 for 20 clients. For the shopping mix it does not improve, but at least it does not worsen like the JDBC-based implementation does. The original version never improves: It remains close to 1 for the read-only mix, and it deteriorates a lot for the other two mixes, most notably for the shopping mix, going down to less than 0.2 of its baseline value.

Considering data from Table 1, again we can see that our solution outperforms the original implementation. This is even more so as the number of clients increases and the values for speed-up grow apart between the two implementations. In Figure 16 we present the performance of the FF-based implementation relative to the JDBC-based implementation. As expected, the relative performance increases with the percentage of read operations. Once more, this is due to the reduction in the cost of database access, especially for read-only transactions. Also, the relative improvement is much higher than in the first test, because the database size is much larger, and so we gain by having all of the data in memory.

Note however, that in our single-server tests we did not take advantage of knowing that there was only one application server running and, therefore, the framework still opened a database connection in the beginning of every memory transaction, as presented in Section 3.6. Nevertheless, in this configuration, the `updateFromChangeLogs` method always returns an empty change log, because the only application server running will, necessarily, be up to date. This means that we could increase even further the performance when running in a single application server environment. In Figure 17, we present this gain for the extreme case of running a read-only mix. Here the application server takes advantage of knowing that it is running a single instance, and it only opens a database connection whenever it needs to load missing data into main memory. During the initial warm-up period, all data eventually becomes loaded and no further database access is needed at all. In this case, we can see that the application is capable of using all of the available hardware parallelism, as the WIPS increase up to the point where the clients reach the number of available cores. This is due to the use of JVSTM, in which read-only transactions do not entail any blocking synchronization whatsoever. If we had kept object versions in persistent storage, then we could also have tested the other mixes with this approach. Unfortunately, the current database structure only keeps the latest version, forcing us to open a database connection at the beginning of each transaction, to ensure consistent reads of any values that may be missing in memory. Finally, Figure 17 corroborates an important underlying assumption in our architecture: Database access takes a heavy toll on the performance of the application server. By reducing this cost, our approach is already capable of greatly increasing throughput, when compared to the original implementation. However, there is still much to gain, if we can further reduce the number of accesses to the database.

Our architecture is better suited for a single application server, running on hardware with many cores and large heaps because, in clustered environments, write transactions cause the invalidation of cache entries in the other application servers, which in turn forces their reloading from the database. Still, it was designed to support multiple application servers sharing the same database. So, we performed

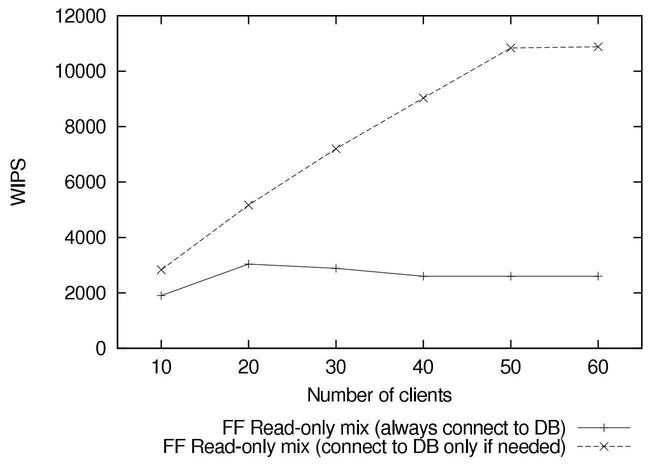

Figure 17. The added cost of database access is clearly seem when comparing two read-only mixes that only differ in whether a database connection is always established at the start of processing a memory transaction.

another test to measure the throughput that could be achieved by increasing the number of application servers for a fixed number of clients. For this final test we used a cluster of ten machines, all connected to the same local network through a 100Mbps switch. Each machine had 2 quad-core Intel Xeon E5506 processors and 8GB of RAM. The versions of Java, MySQL and Tomcat were the same as before. The database was setup in one machine, the clients' simulator in another, and we deployed from one to eight application servers, each in its own machine. The database was populated to support 40 concurrent clients. The clients' simulator always executed 40 clients evenly distributed across the deployed servers. For example, when running a single application server all the 40 clients were directed to that server, whereas when running eight application servers 5 clients were directed at each server.

Figure 18 shows the results obtained. We decided to display the read-only mix in a separate plot, because the WIPS of the FF-based implementation are so much higher than all others that showing them together would make the information harder to read. In the read-only mix, the application servers' caches are never invalidated, because of the absence of writes. For this reason, the throughput increases much more than for other scenarios, as each server is almost independent from the others. The only common aspect is that they all contact the same database.

Considering the other mixes, there is a special condition that occurs in the FF-based implementation for one application server, which causes the WIPS to be close to zero. Here is what happens: The data from the database fits entirely in the application server's memory, taking around 70% of the available memory. Because of the transactional system's infrastructure, update operations take up additional memory in the application server, mostly due to the keeping of multiple versions that may still be needed for some transaction running in an older version. As memory fills up, the garbage collector starts working hard to remove unreachable objects, which include older versions and may include evicting entries from the object cache. Without special tuning of the Java's garbage collector for this particular workload, the default implementation often performs full garbage collections. Once this occurs the whole system degrades. This combination of events leads to a throughput close to zero, as the application server spends most of its time running single-threaded full garbage collections. In the single server scenario, the given rate of write operations combined with memory and garbage collector limitations caused a huge slowdown. In such cases, our implementation will underperform, unless some tuning is applied.

As we split this load by two servers the problem goes away, and we outperform the original implementation, once more. However, the relative performance increase for two or more servers is less noticeable than in the previous single-server tests, because while on the one hand the increase in the number of application servers leads to an increase in parallel processing, on the other hand it also leads to an increase in the number of cache invalidations of each server due to writes by the other servers.

The WIPS for the JDBC-based implementation barely change for any number of servers, in each of the three mixes. In this implementation, most of what the application servers do for each request depends on a database transaction. Adding more servers does not reduce the constant load of 40 concurrent clients requesting database transactions. Thus, we might expect that increasing the number of cores in the database machine would naturally lead to an increase in the system's performance, but that is not necessarily true. If that was the case, then in Figure 15 we should also see some improvement in the JDBC-based implementation as we increase the number of clients, because the machine has 48 cores. In fact, despite the underlying architecture of relational databases having been developed many years ago, their scalability is still a matter of recent discussion [28]. To this contributes the fact that database architectures originated long before computers with many cores became commodity hardware. Despite their long-time focus on efficiently handling multiple concurrent requests, database architectures have also been faced with limited hardware parallelism, which reduces transactions' simultaneous access to its database's internal structures. It has been shown [22] that popular open-source database implementations fail to take advantage of multicore architectures, and simply do not scale as expected.

5. Related Work

Snapshot isolation was first defined in [2]. Today, most database vendors support snapshot isolation as the strongest isolation level (confusingly, also known as serializable in

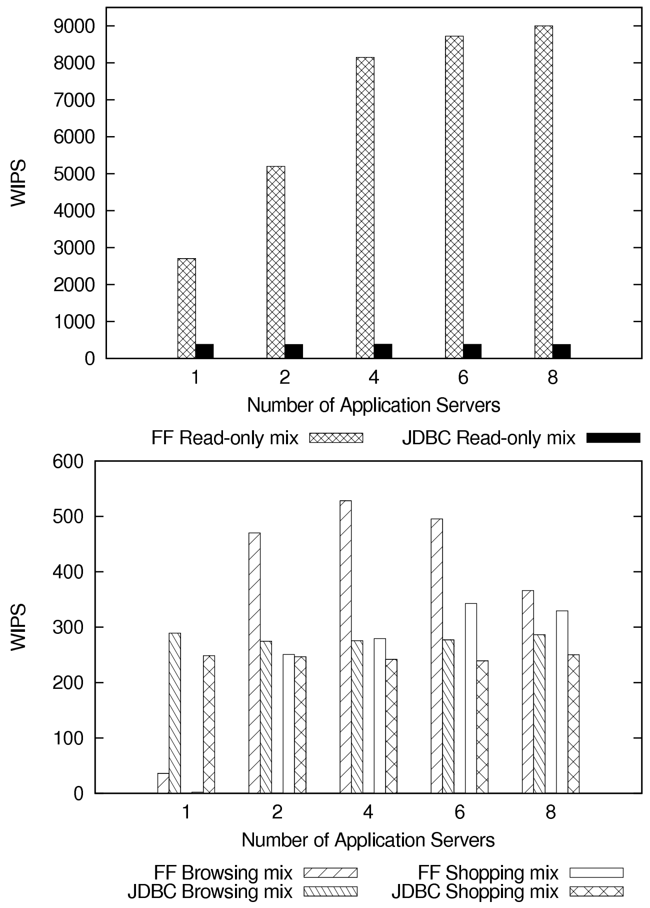

Figure 18. Throughput in a clustered environment for 40 clients evenly split across the servers.

some database implementations), which, despite exhibiting none of the anomalies that the SQL standard prohibits, is not serializable. Since the broad adoption of this isolation level by database implementations, other researchers have cataloged the weaknesses present in this isolation level [12].

In [13] the authors propose a theory to identify when non-serializable executions of applications can occur under snapshot isolation, which has been used to manually demonstrate that some programs may safely run under snapshot isolation without exhibiting any anomalies. This is done, by analyzing possible interferences between concurrent transactions. Additionally, the authors also propose ways to eliminate the identified interferences, but at the cost of manually modifying the program logic of these applications to make it serializable.

Later work [1], has demonstrated that the risks involved in using weaker consistency semantics often do not pay off for the absence of serializability. The effort of ensuring safe executions under snapshot isolation is high, and in many applications the performance penalty for running with full serialization guarantees is low, when compared to the risks of data corruption because of concurrency bugs due to the absence of strong consistency guarantees.

In another work [23] the authors propose an automated tool to detect the snapshot isolation anomalies, but to ensure full coverage, they incur in false positives. Also, computing the minimum set of conflicting transaction pairs that require modification is a NP-Hard problem. Regardless, modifying the program code to eliminate the conflicts is still a manual task.

An alternative to changing program code is to change the transactional engine to ensure strict serializability. This is proposed in [8]. The authors describe an algorithm that provides serializability, while showing, in most cases, a performance similar to the one obtained when running with just snapshot isolation. This work, albeit different from ours, provides evidence that serializability does not incur necessarily in performance problems.

Yet another approach to ensuring program correctness is presented in [3], which defines a new correctness criteria—semantic correctness—that is weaker than serializability, and describes the conditions in which executions preserve given integrity constraints for each isolation level.

Neither of the previously mentioned approaches reduces the number of database queries. Such requires caching on the application server. In [25], the authors present a versioned object cache for J2EE that reduces the number of database accesses and provides snapshot isolation on top of a relational database that must also provide snapshot isolation. They also describe a vertical replication model where each node in the cluster is built from a relational database and a J2EE application server. Their object cache allows for a reduction in the number of database requests, just like ours, but there are other important differences. Most notably, they do not ensure object identity: When an object is modified a new copy is created, which does not allow for object reference comparison. In our implementation, a single object instance is shared among all transactions. Additionally, our transactional system only blocks during the commit of a write transaction whereas they require locking for each object update operation. Most importantly, we ensure strict serializability across the cluster whereas their implementation ensures only snapshot isolation. The focus of their work is on availability and scalability, which led to an architecture of one database per application server, with the cache updates being sent over a dedicated group communication system. However, there is no protocol for dynamically adding a new node to the cluster, which would require database state synchronization to keep up with the rest of the system. In our solution, there is a single point of access to the database—which can be replicated internally—and the cache in each application server is updated from the database, allowing for adding and removing application servers on-the-fly.

Initially, our work concentrated mainly on improving performance for applications running on single computers with many cores. The performance improvements attained in the clustered environment are, in a sense, a by-product of this architecture, which we believe can be even further improved by taking into account the specificities of a distributed system in the design of the system. The work reported in [10, 27], which extends the JVSTM design with scalable and fault-tolerant distributed transactional memory algorithms, is a step in that direction. This work concentrates only on the distribution aspects and, like [25], builds on a group communication system between the nodes, currently implementing only a single entry point to the persistence tier.

6. Conclusion

In this paper we have described a new architecture for the development of enterprise applications, which is based on shifting transaction control from the database to the application server tier. This architectural change was made possible by developments both in hardware as well as software that did not exist when relational databases became widely adopted, thus biasing development towards current architectures. In hardware, the change was enabled by the development of machines with many cores and large memory heaps. In software, STM technology has enabled us to think in terms of transactions managed by the application server and also to a change in the way persistent application objects are managed in main memory.

Some observations led to this change. For one, programmers are often burdened with additional programming efforts because of limitations in the transactional semantics provided by existing systems. By providing strict serializability the programming of concurrent operations becomes simpler and less prone to error. Another problem is related to the cost of database access in current enterprise applications. Often these applications are feature-rich with complex domain and business logic that is implemented in object-oriented applications. These cause several round trips to the persistence tier to transactionally manipulate persistent application data. Often, simply the execution of a read-only operation can trigger several round trips to the persistence tier. The accumulated latency of this communication is no longer negligible and is perceived by the client of the service. Shifting transaction control to the application tier can reduce the need for these accesses with a significant reduction in the time it takes to answer a client's request.

The architecture that we presented is designed to make good use of the improved computational capacity of single application servers running on hardware with many cores and large memory heaps. It excels when applied to object-oriented enterprise applications that have complex object structures and exhibit many concurrent accesses with a high read/write ratio. Based on the evidence that we collected over the past six years, during which we employed this architecture to several applications and collected statistics about their workload patterns, we believe that many of today's mainstream web applications have these same characteristics. Thus, we claim that our architecture is a better solution for many, if not most, of the modern enterprise applications.

It is not a panacea, however, and, as such, is not exempt of problems. For instance, even though our implementation works in clustered environments, it suffers from cache invalidations in scenarios where the rate of write operations is higher. Looking ahead, it is our intention to continue to work in the development of this architecture, trying to broaden the scenarios in which it can be applied to increase application performance and safety, namely in write-heavy scenarios and in continued reduction of the database access costs.

Acknowledgments

The work described in this paper would not have been possible without the contribution of many people that have been involved in some way or another with the development of the Fénix Framework and of the FénixEDU system. These include the excellent development and systems' administration teams at CIIST, and the members of the Software Engineering Group at INESC-ID.

Also, this work was partially supported by FCT, both via INESC-ID multiannual funding through the PIDDAC Program funds and via the Pastramy project (under contract number PTDC/EIA/72405/2006), and, more recently, by the Cloud-TM project, which is co-financed by the European Commission through the contract number 257784.

References

[1] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*, pages 576–585, Cancun, Mexico, 2008. IEEE Computer Society.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*, pages 1–10, San Jose, CA, USA, 1995. ACM.

[3] A. Bernstein, P. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. In *Proceedings of the 16th International Conference on Data Engineering (ICDE '00)*, pages 57–66, San Diego, CA, USA, 2000.

[4] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 251–264, Vancouver, Canada, 2008. ACM.

[5] J. Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Instituto Superior Técnico/Universidade Técnica de Lisboa, Sept. 2007.

[6] J. Cachopo and A. Rito-Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In *Proceedings of the 6th inter-*

national conference on Web engineering (ICWE '06), pages 297–304, Palo Alto, CA, USA, 2006. ACM.

[7] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, Dec. 2006. doi: 10.1016/j.scico.2006.05.009.

[8] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4):20:1–20:42, Dec. 2009. doi: 10.1145/1620585.1620587.

[9] N. Carvalho, J. Cachopo, L. Rodrigues, and A. R. Silva. Versioned transactional shared memory for the FénixEDU web application. In *Proceedings of the 2nd Workshop on Dependable Distributed Data Management (WDDDM '08)*, pages 15–18, Glasgow, Scotland, 2008. ACM.

[10] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*, pages 307–313, Shanghai, China, Nov. 2009.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113, Jan. 2008. doi: 10.1145/1327452.1327492.

[12] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33 (3):12–14, Sept. 2004. doi: 10.1145/1031570.1031573.

[13] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005. doi: 10.1145/1071610.1071615.

[14] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same? *ACM SIGACT News*, 39(1): 48–58, Mar. 2008. ISSN 0163-5700. doi: 10.1145/1360443.1360456.

[15] S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, pages 179–188, San Antonio, TX, USA, 2011. ACM.

[16] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Boston, MA, USA, 2002.

[17] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[18] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 175–184, Salt Lake City, UT, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345233.

[19] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[20] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991. doi: 10.1145/114005.102808.

[21] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A classification of object-relational impedance mismatch. In *Proceedings of the 1st International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA '09)*, pages 36 –43, Cancun, Mexico, Mar. 2009.

[22] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, pages 24–35, Saint Petersburg, Russia, 2009. ACM.

[23] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1263–1274, Vienna, Austria, 2007. VLDB Endowment.

[24] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979. doi: 10.1145/322154.322158.

[25] F. Perez-Sorrosal, M. Patiño-Martinez, R. Jimenez-Peris, and K. Bettina. Consistent and scalable cache replication for multi-tier J2EE applications. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware (Middleware '07)*, pages 328–347, Newport Beach, CA, USA, 2007. Springer-Verlag.

[26] D. Pritchett. BASE: An acid alternative. *Queue*, 6(3):48–55, May 2008. doi: 10.1145/1394127.1394128.

[27] P. Romano, N. Carvalho, M. Couceiro, L. Rodrigues, and J. Cachopo. Towards the integration of distributed transactional memories in application servers' clusters. In *The Third International Workshop on Advanced Architectures and Algorithms for Internet DElivery and Applications (AAA-IDEA '09)*, Las Palmas, Gran Canaria, Nov. 2009. ICST, Springer.

[28] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Communications of the ACM*, 54(6):72–80, June 2011. doi: 10.1145/1953122.1953144.

[29] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, Jan. 2009. doi: 10.1145/1435417.1435432.