TMBean: Optimistic Concurrency in Application Servers Using Transactional Memory

Lucas Charles, Pascal Felber, and Christophe Gête

Computer Science Department, University of Neuchâtel,
Rue Emile-Argand 11, CH-2009 Neuchâtel, Switzerland

Abstract. In this experience report, we present an evaluation of different techniques to manage concurrency in the context of application servers. Traditionally, using entity beans is considered as the only way to synchronize concurrent access to data in Jave EE and using mechanism such as synchronized blocks within EJBs is strongly not recommended. In our evaluation we consider the use of *software transactional memory* to enable concurrent accesses to shared data across different session beans. We are also comparing our approach with using (1) entity beans and (2) session beans synchronized by a global lock.

Keywords: Application Server, Java EE, EJB, Software Transactional Memory.

1 Intoduction

Multicore CPUs have become common nowadays, but taking advantage of their processing power is still considered a hard task. There is a big gap between the scientific community and specialized vendors on one hand, and end developers with domain-specific skills but limited experience with multi-threaded programming on the other hand. In order to help programmers to fully take advantage of multicore architectures, we have to provide them with the right tools that are both efficient and easy to use.

The traditional approach to dealing with concurrency is based on locks. It offers a fine-grained control over the different resources to protect from data races, but has the drawback of being error prone. The programmer faces the trade-off between performance and safety: fine-grained locking is efficient but prone to complex issues like deadlocks or priority inversions, while coarse-grained locking is safe but does not scale.

Another approach to that gained significant attention from the research community is optimistic concurrency control. It relies on the assumption that the number of updates to shared data is by far lower than the number of read accesses, and conflicts are rare. Transactional memory (TM) [9,16] belongs to this category. In transactional memory, critical sections are expressed as atomic blocks performed as *transactions*. At runtime, these transactions may be executed concurrently based on the optimistic expectation that the set of values

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 484–496, 2009.

read by one transaction will not overlap with the set of values written by another concurrent transaction. If no conflict occurs, concurrent transactions can commit, otherwise some of them must rollback and restart their execution. TM has been shown to scale well on multiple cores when the data access pattern behaves "well," i.e., when few conflicts are induced [1,11].

Application servers typically serve multiple clients at the same time. When running on a multicore CPU, they should be able to straightforwardly process concurrent requests from different users. However, it is not trivial to have multiple users access the same stateful application object in a consistent manner without serializing requests. In the context of Java Enterprise Edition (Java EE), while it is common to serve multiple clients accessing the same database, it is far less common to develop applications that allow some concurrent collaboration between Java beans (EJBs).

For several years now, the only way considered safe in a Java EE application to synchronize data among beans, is to use *entity beans*. This approach is using an underlying database which adds significant overhead. It is a common trap for programmers to try to use lock-based synchronization within EJBs, since it is highly error prone, in particular within an ongoing transaction that could abort and produce deadlocks. Using entity beans have the real advantage of providing a certain degree of abstraction, as of how data are shared, but the overhead of the underlying database might be prohibitive if entity beans are solely used for synchronizing concurrent threads. In that case, more lightweight and scalable techniques are desirable, such as those provided by software transactional memory (STM).

In this experience report, we evaluate the use of STM to implement concurrent Java beans in an application server. We present a benchmark in the form of a concurrent Java EE application and we compare different concurrency control strategies based on locks, entity beans and STM. The results support our claim that an STM can provide better scalability than the other strategies while being very simple to use.

The rest of this paper is structured as follows. We discuss related work in Section 2. In Section 3, we describe how concurrency is dealt in the context of a Java EE application. Section 4 explains how optimistic concurrency can be introduced in an application server to take advantage of an existing STM. We describe our benchmark in the Section 5 and discuss experimental results in Section 6. We finally conclude in Section 7.

2 Related Work

While transactional memory has gained much attention recently with multicore machines becoming ubiquitous, it has been first proposed more than a decade ago. Shavit and Touitou described the first STM implantation as a non-blocking mechanism [16], but STM really reached a sufficient level of usability when Herlihy *et al.* proposed DSTM [8], the first STM that was able to manage dynamic data structures. Java support for transactional memory was first introduced by Harris and Fraser [6]. They have modified a Java virtual machine to support the

notion of *atomic blocks*, which as opposed to synchronized blocks are deadlock free and rely upon optimistic concurrency control.

In our evaluation, we use the LSA [13] STM library for Java. It uses a multi-version design (i.e., multiple versions of shared objects are available for increasing the likelihood of successful commits) with a time-based STM algorithm that was shown to be among the most efficient currently known.

Most related to our study, Cachopo *et al.* [2] have also used a multi-version STM to implement a fully distributed Web application. Their application uses STM for local synchronization but relies on a central database to synchronize the data shared among different machines. Results show that STM provides interesting benefits over a database-only approach.

Finally, the latest EJB 3.1 specification introduces a new type of *singleton* session bean that can be accessed concurrently. The approach we present here covers use cases that a singleton session bean should support, the fault tolerance left aside.

3 Application Servers and Concurrent Beans

This section briefly summarizes the architecture of Java EE application servers and describes the different options for executing concurrent beans in such architectures.

3.1 Java EE Architecture

Java EE is a specification of a multi-tier application that provides four different types of components, called "beans", to serve clients requests. On the business side it provides stateless and stateful session beans, while on the storage side it provides entity bean that are representing persistent data maintained in a database. A fourth type, message driven beans, allows asynchronous communication.

Only entity beans may safely be shared by multiple clients. As the clients might change the same data, entity beans often work within transactions. To control the usage of transactions in Java EE, several transactions attributes are defined that indicate whether operations can, must, or cannot be executed within a (new or existing) transaction. When experimenting with STM and lock-based shared session beans, we forbid transactional execution to avoid possible problems, for instance, Java EE transactions aborting while in an atomic block or a critical section.

3.2 Concurrency in Java EE

Java EE recommends that every shared access has to be done through entity beans. This approach has the advantage of lowering considerably the difficulty of concurrency management. By forcing every entity bean to be accessed only within a transaction, it ensures isolation between concurrent accesses and thus provides to the end programmer a high level abstraction of the application synchronization. The major limitation of this approach is the overhead introduced

by the back-end database, especially in case persistence is not necessary and entity beans are only used for synchronization purposes.

Stateless session beans, as defined by the Java EE specification, are not bound to serve the same client. Moreover there is no guarantee that a specific client will always be served by the same bean. Such a loose constraint allows a stateless session bean container to decide how many beans are needed to serve incoming requests. Despite the name "stateless", developers are not prevented from storing values. Nor are they from sharing and accessing data concurrently, within a single Java virtual machine, by declaring static fields. This is the approach we use in our evaluation for experimenting with shared session beans.

In contrast, stateful session beans are bound to serve only one client and they remain alive as long as the client holds a reference to them. Stateful session beans keep a conversational state, which will persist for the duration of the session between the client and the server. A traditional example of such a conversational state is an online shop cart, where selected items are temporarily saved for the duration of the session. The semantics of such beans make them inappropriate for accessing shared data concurrently, and we do not consider them in our evaluation.

While it is discouraged to use synchronized blocks or explicit locks within session beans, they are still of some use when considering multi-threaded scenarios. A major problem with locks is their lack of composability, which could lead for instance to deadlock situations and hang the whole application server if not used properly. In the next section, we present a new approach to overcome this situation based on software transactional memory, which allow us to provide both scalability and composable synchronization.

4 Optimistic Concurrency Using Transactional Memory

In this section, we describe our approach based on transactional memory for executing concurrent beans in a Java EE application server. We first recall the principles underlying STM, then describe the LSA-STM library used in our implementation, and finally discuss how to enable STM within the application server to provide TM-enabled beans.

4.1 Software Transactional Memory

Software transactional memory (STM) [16] has been introduced as a means to support lightweight transactions in concurrent applications. It provides programmers with constructs to delimit transactional operations and implicitly takes care of the correctness of concurrent accesses to shared data. Typically, data accesses that are transactional are tagged by the programmer (or the compiler) and they execute speculatively. For instance, writes are buffered in thread-local storage and they are only committed to shared memory if the transaction completes successfully. In case of conflict, e.g., because two transactions write the same object, a specific component of the STM called *contention manager* decides upon the conflict resolution strategy, which usually implies aborting one of the transactions. Aborted transactions are automatically restarted until they eventually commit.

This type of optimistic concurrency control is particularly appropriate if conflicts are rare and unpredictable, i.e., the use of pessimistic approaches based on locks would limit scalability by serializing critical regions most often unnecessarily. Another important advantage of STM over lock-based synchronization is its composability [7]. STM has been an active field of research over the last few years (e.g., [8,6,10,12,15,3,14,4,11]).

4.2 LSA-STM

LSA-STM is a Java implementation of the *lazy snapshot algorithm* (LSA) [13], a time-based STM algorithm. LSA-STM allows building a consistent view of the objects accessed by the transaction using efficient *invisible* reads (i.e., a read by a transaction is not immediately visible to other transactions and read-write conflicts are detected only at commit time) while avoiding the quadratic cost of incremental validation incurred by previous algorithms (e.g., [12]) to guarantee that transactions have a consistent view of accessed data. LSA builds consistent linearizable snapshot in a very efficient manner. It provides an object-based API, i.e., the granularity of data accesses and conflict detection is a Java object. It can maintain multiple versions of shared objects and guarantees that read-only transactions never need to abort if sufficiently enough object versions are kept. Transaction demarcation is performed at the level of individual methods using Java annotations and accesses to shared objects are instrumented automatically using aspect-oriented programming. LSA-STM is freely available from `http://tmware.org/`, together with a word-based C implementation of LSA (TinySTM) [4].

4.3 TMBeans: STM-Based Concurrent Beans

To provide STM-based concurrency in EJBs, we used stateless session beans augmented with specific transactional objects that represent shared state. These transactional objects are declared as static fields, i.e., they are shared among instances of the beans. This implementation has been realized as a proof of concept, our final goal is to define a new type of beans, *TMBeans*, that would transparently manage shared state and transactional operation executions. The introduction of a new type of bean is following the logic behind the introduction of a singleton bean in the EJB 3.1 specification.

Each time a client invokes an operation on a TMBean, a new STM transaction is started to process the client request. Multiple clients accessing the same TMBean will execute concurrent transaction that may conflict and, hence, abort and restart. If there is no conflict, transactions will execute in parallel and commit. As TMBeans are not persistent and do not use a back-end database, they are expected to have lower overhead than entity beans.

It is important to notice that the transactions used by LSA-STM are not aware of the transactions provided by the application server and thus do not interfere with them. In fact, LSA-STM provides it's own transactions and transactional objects and thus does not use the JTA implementation provided by JBoss.

5 Benchmark

Our evaluation is based on a benchmark which represents a client-server version of a crossword game, where the clients will try to solve concurrently the same grid, i.e., they will pick random words and write their solutions in each of the associated cell.

5.1 The Crossword Benchmark

The game itself was developed as a web application using a client-server architecture in which clients can access the server through a web browser. While this approach allows real users to interact with the application, it is not suitable for testing purpose. Indeed, to load the server we needed a non-interactive application that could a sufficient amount of concurrent requests. To that end, we developed a standard Java application that simulates a group of users. To simplify the benchmark, we assume that words entered by the users are always correct, and we allow entering infinitely many times the same words so that the test can continue even if the grid is complete. The size grid is configurable and the average length of words is function of the grid size.

5.2 Test Application

Figure 1 illustrates the test environment, it mainly consists of two machines connected over a LAN network: a 16-core machine running the JBoss application server that executes the crossword application, and a multi-threaded client application that performs concurrent requests. The server allows clients to enter information in the grid, at the granularity of a word or a single letter. It also gathers various runtime information in order to compute statistics over the different test runs.

As the test focuses on measuring the throughput of the different locking techniques and to avoid the grid to be completed before the end of the experiment, simulated clients do not perform any computation to determine what are the

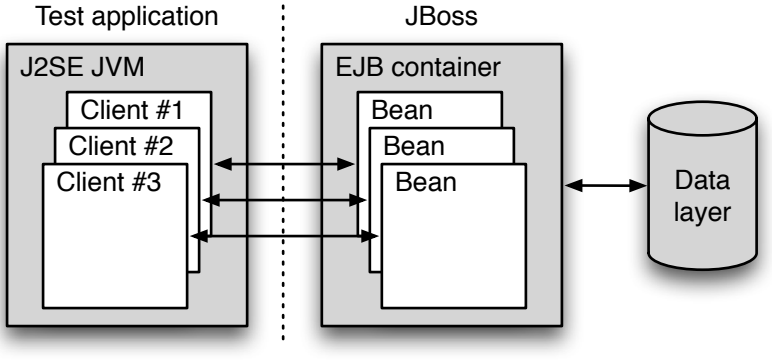

Fig. 1. Overview of the test environment

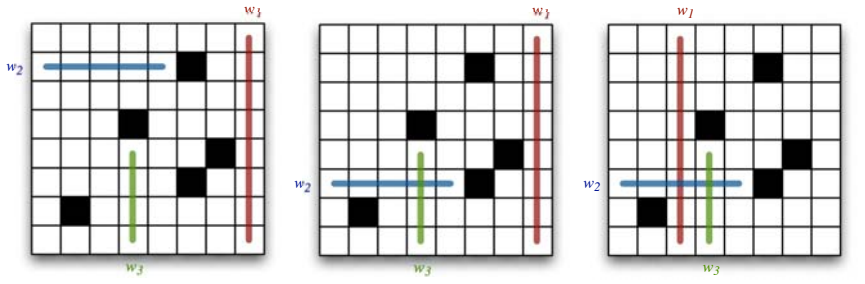

Fig. 2. Three transactions concurrently entering words w_1, w_2, and w_3 in the crossword application. **Left.** Transactions do not conflict and can all commit. **Center.** Transactions entering words w_1 and w_2 conflict and only one can commit. **Right.** All transactions conflict, but aborting the one entering w_2 allows the two others to commit.

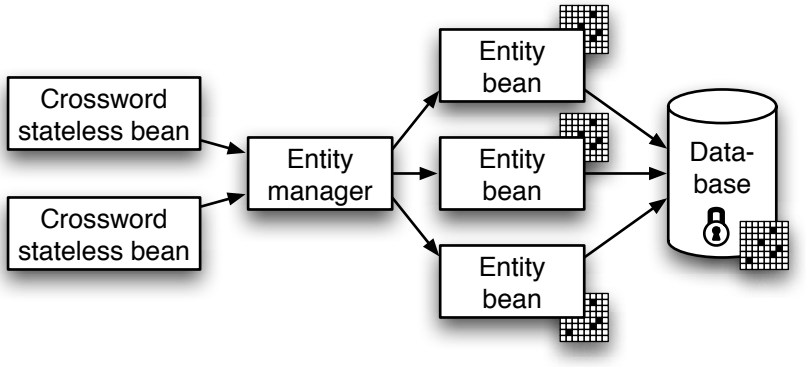

Fig. 3. Architecture overview of the test application using entity beans

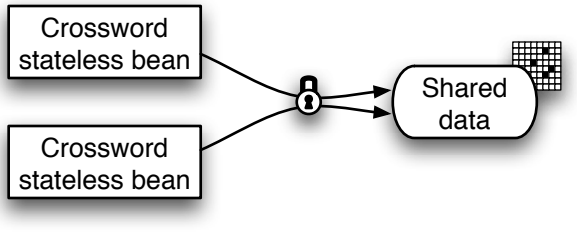

Fig. 4. Architecture overview of the test application using a global lock

suitable words or letters in the grid. Instead the task of a client consists of three steps. First, it chooses randomly a cell in the grid and a direction. Second, it reads the selected word in the grid, which implies a read access in the transaction. Finally, either the word is not yet in the grid and the client writes it, or

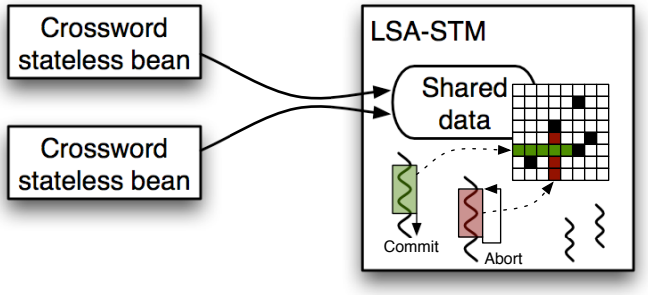

Fig. 5. Architecture overview of the test application using an STM

the word is already present and the client erases it. Therefore, in every case a client performs a read followed by a write.

As shown in Figure 2, several transactions accessing the same cell in the grid conflict and one has to abort and restart. When more than two transactions conflict, it is desirable to abort as few transactions as possible to let other complete their execution (see the right side of the figure). This choice is difficult to take in practice as contention managers only deal with pairwise conflicts. In our implementation we use the *greedy* contention management policy [5]. Note that, to simplify the procedure, we consider two write accesses to the same cell as conflicting even if they write the same letter.

5.3 Concurrency Control Strategies

We evaluate three concurrency control strategies in our tests. First, we use entity beans to orchestrate simultaneous requests from the clients (see Figure 3). Second, we use a simple approach with a global lock that protects the shared grid, stored as a static field of a stateless session bean class (see Figure 4). Finally, we evaluate the TMBeans approach, with a shared object protexted by STM transactions executed by multiple stateless session beans (see Figure 5).

5.4 Experimental Setup

Our experiment has been conducted on the server side with four quad-core AMD Opteron processors (16 cores) clocked at 2.20 GHz equipped with 8 GB of ram. The environnment used was running JBoss 4.0.5 on top of openSUSE 10.3. On the client side, we used a dual-core Intel processor clocked at 2.80 Ghz equipped with 2 GB of ram running Windows XP. The fact that a single machine hosts all the clients has no impact, as the network latency dominates the overhead of context switches on the client.

Three different criteria influence the degree of contention during a test. First obviously, the number of clients accessing the server: the more concurrent accesses we have, the higher the contention will be. Second, the size of the grid, since the probability that two transactions conflicts, i.e. pick two words with at

least one letter in common, is lower with a larger grid. Third, the network latency being more important than the cost of in memory operations, every request will be processed far quicker than the round-trip-time of a message to the server; thus a transaction is likely to commit before a conflicting request from another client reaches the server. In order to circumvent this third point and highlight the benefits of optimistic concurrency, we introduced a configurable delay between each letters of a word entered in the grid. This allows us to simulate longer transactions and increase the likelihood of conflicts.

6 Results

In this section we compare the results we obtained from our benchmark. The results were reported by the clients at the end of every test run. The duration of each run is one minute and the graphs represent the average of 5 runs.

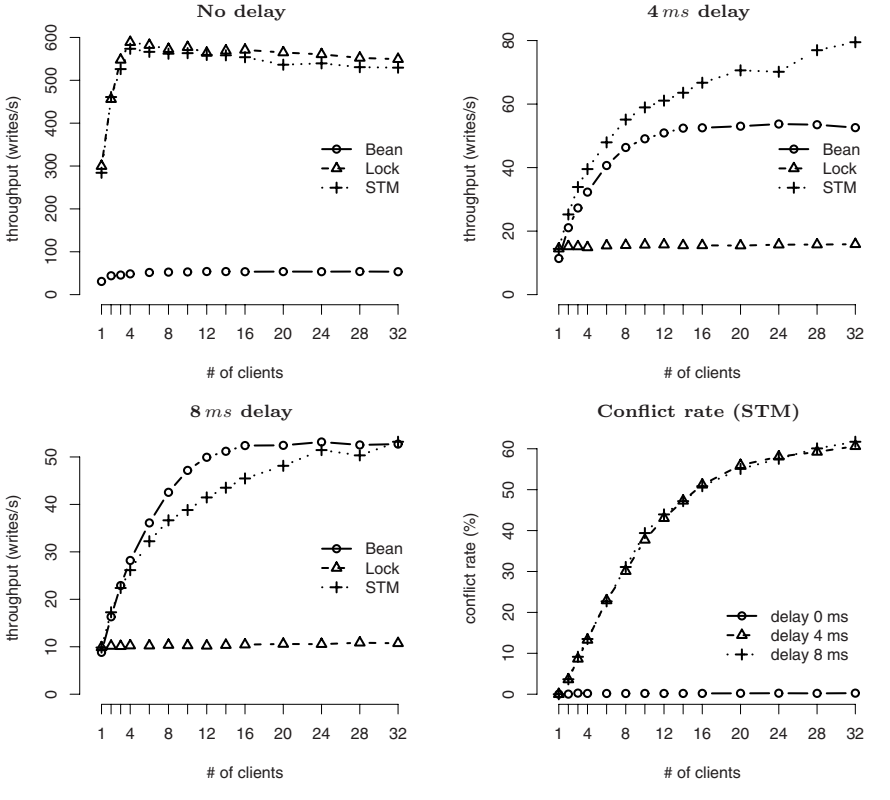

Fig. 6. Influence of the delay under high contention, with a 10x10 grid. **Top left.** No delay. **Top right.** 4 *ms* delay between letters. **Bottom left.** 8 *ms* delay between letters. **Bottom right.** Conflict rate for the STM strategy.

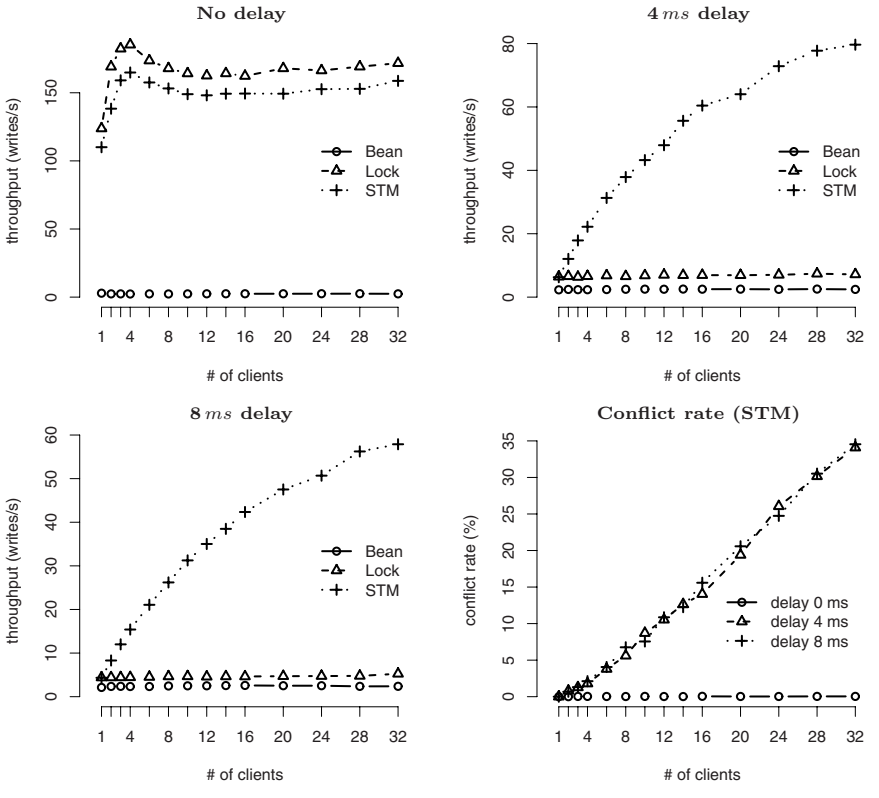

Fig. 7. Influence of the delay under low contention, with a 50x50 grid. **Top left.** No delay. **Top right.** 4 *ms* delay between letters. **Bottom.** 8 *ms* delay between letters. **Bottom right.** Conflict rate for the STM strategy.

6.1 Influence of Transaction Duration

As mentioned above, the role of the delay is to avoid the serialization effect the network has on parallel requests. Figure 6 shows the performance of the three concurrency control strategies on a small grid (10x10) with delays of 0, 4, and 8 *ms* and a variable number of clients. With no delay, and thus almost no contention as latency of the network is much higher than the processing of a transaction, we observe that the STM-based and the lock-based implementations perform similarly (although the latter seems to be slightly faster than the former because it does not incur the cost of transactional memory accesses). Both approaches are efficient as they only write to memory. In contrast, entity beans have a significantly lower throughput due to the cost of accesses to the database.

When increasing the delay to 4 *ms*, unsurprisingly the global lock strategy does not scale well. STM performs best with entity beans a close second, because the delay dominates the cost of database accesses. The trends are similar with 8 *ms*

delays, but this time entity beans perform slightly better than STM. This can be explained by the fact that, under high contention, many STM transactions will abort and restart, thus reducing the overall throughput. Despite this limitation, we observe that the STM strategy still scales well up to 32 concurrent clients.

To validate our observations, we have observed the conflict rate for the STM strategy with the three delay values used in our tests (see Figure 6, bottom right). We observe that with no delay, there are almost no conflicts independently of the number of clients as transactions are much shorter than network communications. With delays of 4 and 8 *ms*, the conflict rate increases up to 60%, i.e., more than half of the transactions abort once before completing successfully.

Figure 7 shows the same data but for a 50x50 grid, i.e., with less contention. In comparison to the small grid, we observe here that entity beans perform poorly with all delay values. This can be explained by the fact that larger grids have longer words on average, and the per-letter access cost of entity beans impairs scalability. We also observe that scalability of the STM strategy is better (closer to linear), and the conflict rate remains below 35%.

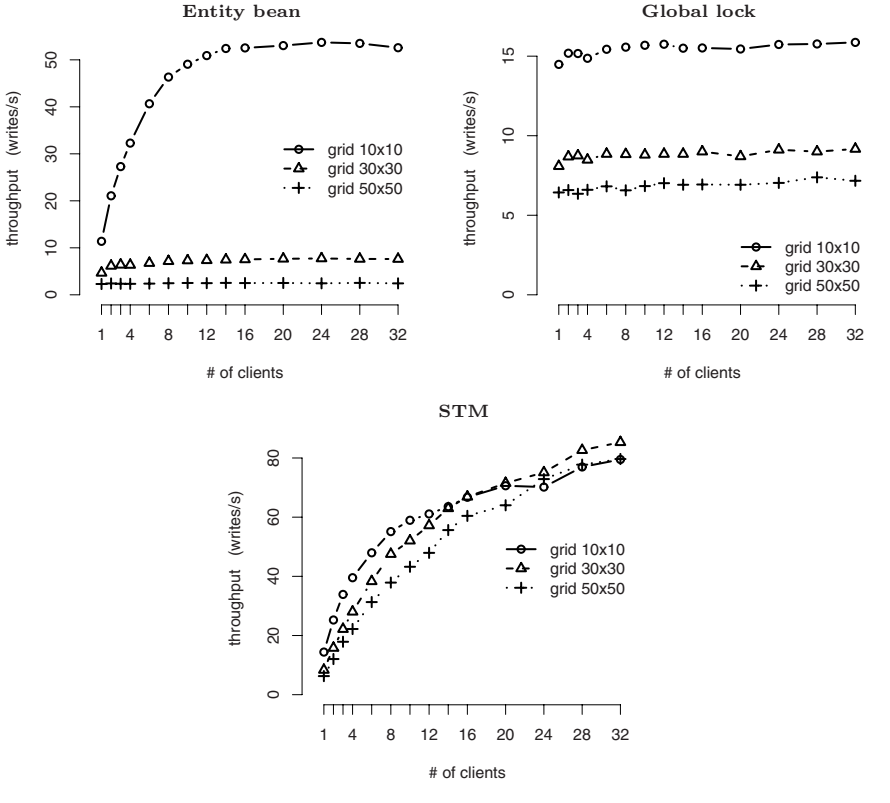

Fig. 8. Influence of the grid size with 4 *ms* delay between letters. **Top left.** Entity bean. **Top right.** Global lock. **Bottom.** STM.

6.2 Influence of Grid Sizes

The size of the grid has an important impact on the throughput as it influences both the probability for two transactions to conflict and the amount of data to be written (as words will be longer on average). In Figure 8, we compare three grid sizes (10x10, 30x30, 50x50) with a $4\,ms$ delay between entering the letters of a word. One can first observe that the STM strategy performs noticeably better than the other approaches with all configurations. Entity beans scale reasonably well with small grids, but throughput quickly saturates with large grids. The performance of the lock-based approach remains flat as client requests are serialized. Only the STM strategy scales well independently of the grid size.

7 Conclusion

In this experience report, we have presented an evaluation of two traditional techniques for managing concurrency in the context of Java EE Web applications—entity beans and coarse grained locking and compared them with a novel technique that uses software transactional memory. Our benchmark allowed us to pinpoint the benefits of using optimistic concurrency control in scenarios where conflicts are rare. In most of our experiments, STM-based beans showed better performance and scalability than the other approaches. These results demonstrate the potential benefits of providing a new form of STM-enabled beans, *TMBeans*, in application servers to enable programmers to develop lightweight concurrent stateful beans.

Acknowledgements. This work was supported in part by European Union grant FP7-ICT-2007-1 (project VELOX).

References

1. Adl-Tabatabai, A.-R., Kozyrakis, C., Saha, B.: Unlocking concurrency. Queue 4(10), 24–33 (2007)
2. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. In: Proceedings of SCOOL (2005)
3. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proceedings of DISC (September 2006)
4. Felber, P., Riegel, T., Fetzer, C.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of PPoPP (February 2008)
5. Guerraoui, R., Herlihy, M., Pochon, S.: Toward a theory of transactional contention managers. In: Proceedings of PODC (July 2005)
6. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of OOPSLA (October 2003)
7. Harris, T., Herlihy, M., Marlow, S., Peyton-Jones, S.: Composable memory transactions. In: Proceedings of PPoPP (June 2005)
8. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of PODC (July 2003)

9. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of ISCA (1993)
10. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of PODC (July 2005)
11. Larus, J., Kozyrakis, C.: Transactional memory. Communication of the ACM 51(7), 80–88 (2008)
12. Marathe, V.J., Scherer III, W.N., Scott, M.L.: Adaptive software transactional memory. In: Proceedings of DISC (2005)
13. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proceedings of DISC (September 2006)
14. Riegel, T., Fetzer, C., Felber, P.: Time-based transactional memory with scalable time bases. In: Proceedings of SPAA (June 2007)
15. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of PPoPP (2006)
16. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)