Concurrent Byzantine Fault Tolerance for Software-Transactional-Memory Based Applications

Honglei Zhang and Wenbing Zhao

***Abstract*—Typical Byzantine fault tolerance algorithms require the application requests to be executed sequentially, which may severely limit the throughput of the system considering that modern CPUs are equipped with multiple processing cores. In this paper, we present the design and implementation of a Byzantine fault tolerance framework for software-transactional-memory based applications that aims to maximize concurrent processing while preserving strong replica consistency. The approach is based on the idea of committing concurrent transactions according to the total order of the requests that triggered the transactions. A comprehensive performance evaluation is carried out to characterize the effectiveness and limitations of this approach.**

***Index Terms*—Byzantine fault tolerance, software transactional memory, distributed systems, concurrent computing, performance evaluation**

I. Introduction

Byzantine fault tolerance (BFT) [1], [2] appears to be a powerful technology to enhance the trustworthiness of distributed applications. In the past decade, we have seen significant advancement of both the efficiency and robustness of BFT algorithms [1]–[5]. However, typical BFT algorithms require the application requests to be executed sequentially, which may severely limit the throughput of the system considering that modern CPUs are equipped with multiple processing cores. This issue has been addressed by a number of researchers [3], [4], [6], [7]. The primary approach is to enable concurrent execution of requests that do no involve conflicting operations. However, to enable concurrent execution, it is assumed that the application semantics is already known. This inevitably increases the design cost of such BFT solutions and limits their reusability for other applications.

In this paper, we present the design and implementation of a BFT framework for software-transactional-memory based applications that aims to maximize concurrent processing without requiring the knowledge of application semantics. The approach is based on the idea of committing concurrent transactions according to the total order of the requests that triggered the transactions. In essence, the dependency between different requests is discovered dynamically (and automatically) by the software-transactional-memory runtime. Non-conflicting requests can be processed concurrently with the only constraint that the commit order must respect the total order of the corresponding requests. Some of the conflicting requests that have been processed concurrently may have to be aborted and retried. A comprehensive performance evaluation is carried out to characterize the effectiveness and limitations of this approach.

Manuscript received April 10, 2012; revised May 8, 2012.

Honglei Zhang and Wenbing Zhao are with the Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH 44115, USA (e-mail: wenbing@ieee.org.)

II. Related Work

The primary approach to increasing the performance of BFT systems is by exploiting application semantics. In PBFT [1], Castro and Liskov noted that read-only requests can be delivered without the need of total ordering. In BASE [2], it was recognized that a BFT system can be made more robust (to minimize deterministic software errors) by adopting a common abstract specification for the service to be replicated. A conformance wrapper for each distinct implementation is then developed to ensure that it behaves according to the common specification. Furthermore, an abstraction function and one of its inverses are needed to map between the concrete state of each implementation and the common abstract state.

In [3], Kotla and Dahlin proposed to exploit application semantics for higher throughput by parallelizing the execution of independent requests. They outlined a method to determine if a request is dependent on any pending request using application specific rules. In [4], Distler and Kapitza further extended Kotla and Dahlin's work by introducing a scheme to execute a request on only a selected subset of replicas. This scheme assumes that the state variables accessed by each request are known, and that the state object distribution and object access are uniform.

In prior work [6], [7], we proposed to rely on deeper application semantics to not only enable more requests (such as those that are commutative) to be executed concurrently, but also minimize the number of Byzantine agreement steps used in an application (particularly for session-oriented applications).

This paper takes a drastically different approach from those mentioned above. Rather than resorting to the application semantics, which may be expensive to acquire accurately and hard to reuse, we rely on the use of software transactional memory to dynamically capture the dependency of concurrent operations automatically. This approach is inspired by the work of Brito, Fetzer, and Felber [8], where a similar idea was used to ensure multithreaded execution for actively-replicated event stream processing systems. Our work applies the idea in a different context (i.e., Byzantine fault tolerance instead of crash fault tolerance) and furthermore, we carry out detailed experiments and analysis on the level of concurrency that can

be achieved under various conditions.

III. Concurrent BFT Framework

The proposed Byzantine fault tolerance framework is shown in Figure 1. The framework supports client-server applications where the server is constructed using software transactional memory. In our implementation, the LSA-STM open source library [9] is used to enable software transactional memory. The total ordering of the requests sent by the clients is ensured by using the UpRight agreement cluster [5]. The replicated server is running within a separate cluster. Due to the separation of agreement and execution [10], only 2f + 1 server replicas are needed to tolerate up to f faulty replicas.

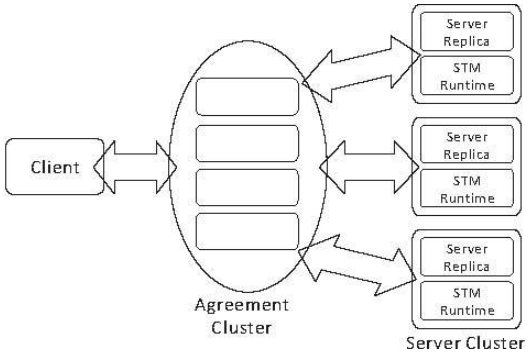

Fig. 1. The proposed Byzantine fault tolerance framework.

The agreement cluster dispatches totally ordered requests (from the clients) to the server replicas. At the server replica, we assume that each request triggers one and only one transaction. A deterministic algorithm is used to assign a multi-dimensional monotonically increasing timestamp to each request and the corresponding transaction (the sequence number assigned by the agreement cluster cannot be directly used because of batching, i.e., multiple requests in the same batch are assigned the same sequence number). The timestamp is then used to ensure the total ordering of the commit of the transactions.

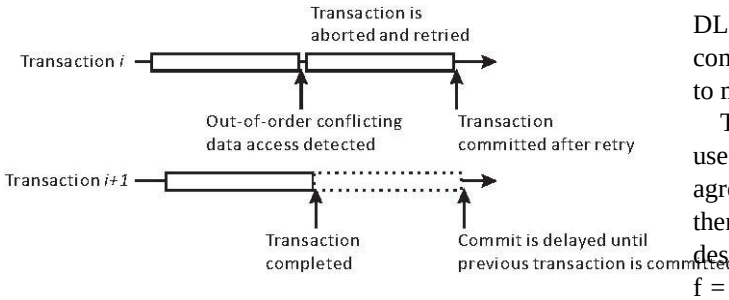

Fig. 2. A transaction is aborted and retried if it accessed shared data items in a conflicting operation out-of-order.

A request is delivered immediately at each replica once it is known that it has been totally ordered. The request is handled by one of the threads in a pre-allocated thread pool. This approach could significantly increase the throughput in systems equipped with multi-core processors. In general, the number of concurrent transactions (i.e., the size of the thread pool) should equal the number of CPU cores. It is possible that a transaction accesses a data item out of order (e.g., transaction i reads a shared data item, and then transaction i − 1, which is ordered ahead of transaction i, later writes to the same shared data item), in which case, transaction i is aborted and retried as soon as the out-of-order conflicting operation is detected, as shown in Figure 2. It is important to note that all transactions ordered later (such as transaction i + 1) would have to wait until the retried transaction has been committed.

IV. Implementation and Performance Evaluation

The proposed Byzantine fault tolerance framework is implemented in Java based on LSA-STM [9] and UpRight [5]. A comprehensive experimental study has been carried out on the research prototype in a Local-Area Network testbed that consists of 14 HP BL460c blade servers and 18 HP ProLiant DL320 G6 rack-mounted servers connected by a Cisco Gigabit switch. Each BL460c server is equipped with two Xeon E5405 (2GHz) processors and 5GB RAM. Each DL320 server is equipped with one Xeon E5620 (2.40GHz) processor and 8GB RAM. All servers run the 64-bit Ubuntu server Linux operating system.

The test application is a client-server application where the server is written using LSA-STM. Each client's request triggers a new transaction at the server. If a transaction is aborted, it will be retried until it is committed. The server maintains a shared data pool that consists of 100 data items. Each transaction accesses 10 data items (all of them are write operations for simplicity). A transaction accesses the data items in the shared data pool pseudo-randomly according to a predefined sharing rate. For example, a 20% sharing rate means that a transaction will access 2 data items in the shared data pool and 8 private data items. To characterize non-trivial processing load, a finite processing delay is artificially introduced at the server for each transaction in the form of busy loops (i.e., the server executes a while loop until the predefined time has passed). Two types of processing load are experimented: (1) fixed at 5ms, and (2) random processing delays with a Poisson distribution with a mean of 5ms.

In the test, the server replicas are deployed among the BL460c blade servers, and the clients are deployed among the DL320 servers. A thread pool of 8 threads is used to enable concurrent processing of up to 8 requests at the server (this is to match the 8 CPU cores at each server node).

The server is replicated with f =1 (i.e., 3 server replicas are used). The client sends a request first to the UpRight agreement cluster for total ordering. The agreement cluster then forwards the request to the server replicas with the designated total order for processing. In the agreement cluster, f = 1 is also used (i.e., 4 agreement nodes are used) for the Byzantine agreement on the total order of requests.

During the experiments, the following scenarios are tested:

1) Fixed processing time (5ms) for each transaction in our BFT framework, denoted as C-BFT (Fixed-i%) in the test result figures, where i is the data sharing rate;
2) Random processing time with Poisson distribution with a mean of 5ms for each transaction in our BFT framework, denoted as C-BFT (Poisson-i%) in the test results figures;
3) For comparison, concurrent processing is disabled (i.e., all requests are processed sequentially one after another), denoted as S-BFT in the test result figures.

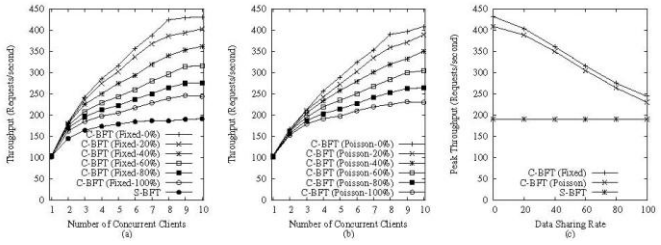

Fig. 3. Test results. (a) Throughput versus the number of concurrent clients for C-BFT Fixed configurations. For comparison, the throughput for S-BFT is also included. (b) Throughput versus the number of concurrent clients for C-BFT Poisson configurations. (c) Peak throughput versus different data sharing rates.

The throughput test results are summarized in Figure 3. Figure 3(a) shows the average throughput with respect to different number of concurrent clients under various C-BFT Fixed scenarios, and the S-BFT scenario. The results for C-BFT Poisson scenarios are shown in Figure 3(b). As expected, the lowest throughput is observed for the S-BFT configuration and the highest throughput is achieved under C-BFT (Fixed-0%) (i.e., when there is no shared data between different transactions). Figure 3(c) shows the peak throughput dependency on the data sharing rate for the three sets of scenarios.

To study the inner workings of the system, the number of conflicts and aborts are profiled, in addition to the number of commits in each run. The profiling results for C-BFT Fixed scenarios are shown in Figure 4 (the results for C-BFT Poisson scenarios are very similar, and hence, they are omitted in the figure). It can be seen that the conflict and abort rates increase exponentially with the number of concurrent clients, and with the sharing rate.

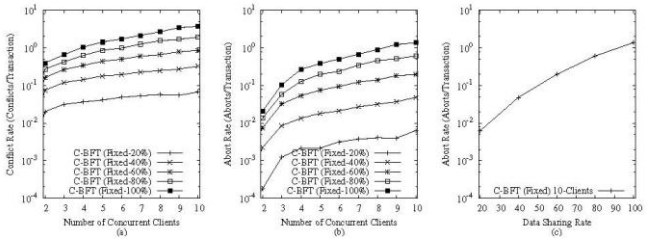

Fig. 4. Conflict and abort rates for C-BFT Fixed. (a) Conflict rate in terms of average number of conflicts per transaction versus different number of concurrent clients. (b) Abort rate in terms of average number of aborts per transaction versus different number of concurrent clients. (c) Abort rates observed for 10 concurrent clients with different data sharing rates.

The test results shown above prove that indeed the throughput is improved with the proposed system compared with sequential BFT processing (i.e., the S-BFT scenario) in all circumstances tested. The throughput improvement ranges from about 28% (with 100% sharing rate), to 125% (with 0% sharing rate). We can make the following two observations from the test results:

1) The throughput is higher with smaller data sharing rate, and
2) The throughput is higher with a uniform fixed processing time for each transaction (i.e., each transaction takes identical fixed amount of time to complete).

Both observations can be easily explained. The higher the data sharing rate, the more likely some transactions will be aborted and retried, as illustrated in Figure 4. Furthermore, when a transaction is retried, all transactions ordered after this transaction may have to wait before they can be committed. This explains observation 1. When all transactions take the same amount of time to complete, the next transaction can be committed almost immediately after the current one is committed, which minimizes any potential wait-to-commit time. On the other hand, if the processing time for each transaction is randomly distributed, it is very likely some transactions will have to wait before they can be committed, which reduces the throughput. This explains the observation 2.

One might expect a much sharper reduction in throughput with the increase of number of concurrent clients and sharing rate due to the observed exponential increase in conflict and abort rates. This did not happen because the aborted transactions can be retried concurrently as well.

Furthermore, the test results also reveal that the proposed system could be further improved. When 0% sharing rate is used, the peak throughput is only about 2.3 times that of sequential BFT. In an ideal scalable system, the peak throughput would be 8 times that of sequential BFT. The less than ideal scalability of the proposed system may be partially due to the restriction of the total ordering of the commits. It is possible to relax this restriction by incorporating the knowledge of application semantics.

V. Conclusion

In this paper, we described a concurrent BFT framework for applications based on software transactional memory. We have done extensive performance evaluation of the proposed framework. The results show indeed the throughput is increased significantly compared with sequential BFT, even in the worst case when every transaction accesses data from the shared data pool. We observed that the throughput strongly depends on the data sharing rate among the transactions. Furthermore, the distribution of processing time of the transactions also plays a role in determining the average throughput. Better throughput can be achieved if all transactions take similar amount of time to complete.

Acknowledgments

This work was supported in part by NSF grant CNS 0821319, by a CSUSI grant, and by a Doctoral Dissertation Research Expense Award (for the first author) from Cleveland State University.

References

[1] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems* 2002, vol. 20, pp.398–461.

[2] M. Castro, R. Rodrigues, B. Liskov, “Base: Using abstraction to improve fault tolerance,” *ACM Transactions on Computer Systems* 2003, vol. 21, pp. 236–269.

[3] R. Kotlan and M. Dahlin, “High throughput byzantine fault tolerance,” *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

[4] T. Distler and R. Kapitza, “Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency,” *Proceedings of the sixth Eurosys conference*, 2011.

[5] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 277–290, 2009.

[6] H. Zhang, H. Chai, W. Zhao, P. M. Melliar-Smith, L. E. Moser, " Trustworthy coordination for web service atomic transactions," *IEEE Transactions on Parallel and Distributed Systems* (to appear) 2012.

[7] H. Chai, H. Zhang, W. Zhao, P. M. Melliar-Smith, L. E. Moser. Toward trustworthy coordination for web service business activities. *IEEE Transactions on Services Computing* (to appear), 2012.

[8] A. Brito, C. Fetzer, and P. Felber, "Multithreading-enabled active replication for event stream processing operators," in *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, 2009, pp. 22–31.

[9] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th International Symposium on Distributed Computing*, 2006, pp. 284–298.

[10] J. Yin, J. P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *Proceedings of the ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003, pp. 253–267.