Logging Last Resource Optimization for Distributed Transactions in Oracle WebLogic Server

Tom Barnes Adam Messinger Paul Parkinson Amit Ganesh German Shegalov
Saraswathy Narayan Srinivas Kareenhalli

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065, USA

{firstname.lastname}@oracle.com

ABSTRACT

State-of-the-art OLTP systems execute distributed transactions using XA-2PC protocol, a presumed-abort variant of the Two-Phase Commit (2PC) protocol. While the XA specification provides for the Read-Only and 1PC optimizations of 2PC, it does not deal with another important optimization, coined Nested 2PC. In this paper, we describe the Logging Last Resource (LLR) optimization in Oracle WebLogic Server (WLS). It adapts and improves the Nested 2PC optimization to/for the Java Enterprise Edition (JEE) environment. It allows reducing the number of *forced* (synchronous) writes and the number of exchanged messages when executing distributed transactions that span multiple transactional resources including a SQL database integrated as a JDBC datasource. This optimization has been validated in SPECjAppServer2004 (a standard industry benchmark for JEE) and a variety of internal benchmarks. LLR has been successfully deployed by high-profile customers in mission-critical high-performance applications.

1. INTRODUCTION

A *transaction* (a sequence of operations delimited by *commit* or *rollback* calls) is a so-called ACID contract between a client application and a transactional resource such as a database or a messaging system that guarantees: 1) **Atomicity**: effects of aborted transactions (hit by a failure prior to commit or explicitly aborted by the user via *rollback*) are erased. 2) **Consistency**: transactions violating consistency constraints are automatically rejected/aborted. 3) **Isolation**: from the application perspective, transactions are executed one at a time even in typical multi-user deployments. 4) **Durability (Persistence)**: state modifications by committed transactions survive subsequent system failures (i.e., redone when necessary) [13].

Transactions are usually implemented using a sequential recovery log containing *undo* and *redo* information concluded by a *commit record* stored in persistent memory such as a hard disk. A transaction is considered committed by recovery when its commit record is present in the log. When a transaction involves several resources (*participants*, also denoted as *agents* in the literature) with separate recovery logs (regardless whether local or remote), the commit process has to be coordinated in order to prevent inconsistent subtransaction outcomes. A dedicated resource or one of the participants is chosen to coordinate the transaction. To avoid inconsistent subtransaction outcomes, the *transaction coordinator* (TC) executes a client commit request using a 2PC protocol. Several *presume-nothing*, *presumed-abort*, and *presumed-commit* variants of 2PC are known in the literature [2, 3, 4, 5, 13]. We briefly outline the Presumed-Abort 2PC (PA2PC) because it has been chosen to implement the XA standard [12] that is predominant in today's OLTP world.

EDBT 2010, March 22-26, 2010, Lausanne, Switzerland.

1.1 Presumed-Abort 2PC

Voting (Prepare) Phase: TC sends a *prepare* message to every participant. When the participant determines that its subtransaction can be committed, it makes subtransaction recoverable and replies with a positive vote (*ACK*). Subtransaction recoverability is achieved by creating a special *prepared* log record and forcing the transaction log to disk. Since the transaction is not committed yet, no locks are released to ensure the chosen isolation level. When the participant determines that the transaction is not committable for whatever reason, the transaction is aborted; nothing has to be force-logged (presumed abort); a negative vote is returned (*NACK*).

Commit Phase: When every participant returned an *ACK*, the TC force-logs the *committed* record for the current transaction, and notifies all participants using *commit* messages. Upon receiving a *commit* message, participants force-log the commit decision, release locks (acquired for transaction isolation), and send a commit status to the TC. Upon collecting all commit status messages from the participants, the TC can discard the transaction information (i.e., release log records for garbage collection).

Abort Phase: When at least one participant sends a *NACK*, the TC sends rollback messages without forcing the log (presumed abort) and does not have to wait for rollback status messages.

The complexity of a failure-free run of a PA2PC transaction on n participants accumulates to *2n+1* forced writes and *4n* messages. Since commit/rollback status messages are needed only for the garbage collection at the coordinator site, they can be sent asynchronously, e.g., as a batch, or they can be piggybacked on the vote messages of the next transaction instance. Thus, the communication cost is reduced to *3n* messages, and more importantly to just *1* synchronous round trip as seen above. If one of the partici-

```
void processess(TwoPCMessage msg) {
  switch (msg) {
  case PREPARE:
    if (i == n) {
      if (executeCommit() == OK) {
        sendCommitTo(msg.from);
      } else {
        executeAbort();
        sendAbortTo(msg.from);
      }
    } else {
      if (executePrepare() == OK) {
        sendPrepareTo(i+1);
      } else {
        executeAbort();
        sendAbortTo(i-1);
        sendAbortTo(i+1);
      }
    }
    break;
  case ABORT:
    if (i != n) {
      sendAbortTo(msg.from == i-1 ? i+1 : i-1);
    }
    executeAbort();
    break;
  case COMMIT:
    if (i != 1) {
      sendCommitTo(i-1);
    }
    executeCommit();
    break;
  } // switch
}

// a)normal operation
//  ->prepare->  ->prepare->  ->prepare->
// P1          P2           P3           P4
//  <-commit<--  <--commit<-  <--commit<-
//
// b) failure on P3
//  ->prepare->  ->prepare->  -->abort-->
// P1          P2           P3          P4
//  <--abort<--  <--abort<--
```

Figure 1: C-style pseudo code of Nested 2PC, and its effect in a commit (a) and an abort (b) case.

pants takes the role of the coordinator, we save one set of messages at the coordinator site, and we save one forced write since we do not need to log transaction commit twice at the coordinator site. Thus, the overhead of a PA2PC commit can be further reduced to *2n* forced writes and *3(n-1)* messages [13].

1.2 Related Work

There are several well-known optimizations of the 2PC protocol. The following two optimizations are described in the XA spec and *should* be implemented by the vendors:

Read-Only Optimization: A participant may vote "read-only" indicating that no updates have been made on this participant on behalf of the current transaction. Thus, this participant should be skipped during the second phase.

One-Phase Commit (1PC): The TC should detect situations in which only one participant is enlisted in a distributed transaction and skip the vote phase.

Nested 2PC: Another interesting optimization that is not part of the XA standard has originally been described by Jim Gray in [2]. It is also known as *Tree* or *Recursive* 2PC. It assumes that we deal with a fixed linear topology of participants P_1, …, P_n. There is no fixed TC. Instead the role of TC is propagated left-to-right during the vote phase and right-to-left during the commit phase. Clients call *commit_2pc* on P_1 that subsequently generates a *prepare* message to itself. Each P_i (indicated by variable *i* in the pseudo-code) executes the simplified logic outlined in Figure 1 processing a 2PC message from the source P_{from}. A failure-free execution of an instance of the Nested 2PC Optimization costs *2n* forced writes and *2*(*n-1*) messages.

1.3 Java Transaction API

Java Transaction API (JTA) is a Java "translation" of the XA specification; it defines standard Java interfaces between a TC and the parties involved in a distributed transaction system: the transactional resources (e.g., messaging and database systems), the application server, and the transactional client applications [11]. A TC implements the JTA interface *javax.transaction.TransactionManager*. Conversely, in order to participate in 2PC transactions, participant resources implement the JTA interface *javax.transaction.xa.XAResource*. In order to support a hierarchical 2PC, a TC also implements *javax.transaction.xa.XAResource* that can be enlisted by third-party TC's. From this point on, we will use the JTA jargon only: we say transaction manager (TM) for a 2PC TC, and (XA)Resource for a 2PC participant.

Every transaction has a unique id that is encoded into a structure named *Xid* (along with the id of the TM) representing a subtransaction or a branch in the XA jargon. The *isSameRM* method allows the TM to compare the resource it is about to enlist in the current transaction against every resource it has already enlisted to reduce the complexity of 2PC during the commit processing, and potentially enable 1PC Optimization. The TM makes sure that the work done on behalf of a transaction is delimited with the calls to *start(xid)* and *end(xid)*, respectively, which adds at least *4n* messages per transaction since these calls are typically blocking. When a client requests a commit of a transaction, the TM first calls prepare on every enlisted resource and based on the returned statuses, in the second phase, it calls either commit or rollback. After a crash, the TM contacts all registered *XAReresource*'s and executes a misnamed method *recover* to obtain the Xid list of prepared but not committed, so-called in-doubt transactions. The TM then checks every *Xid* belonging to it (there can be other TM's *Xid*'s) against its log and (re)issues the *commit* call if the commit entry is present in the log; otherwise *abort* is called.

1.4 Contribution

Our contribution is an implementation of the LLR optimization through JDBC datasources in WLS [8], an adaptation of the Nested 2PC optimization to the JEE. The rest of the paper is organized as follows: Section 2 outlines relevant WLS components, Section 3 discusses the design considerations of the LLR optimization, Section 4 describes the implementation details of normal operation and failure recovery, Section 5 highlights performance achieved in a standard JEE benchmark, and finally Section 6 concludes this paper.

2. ORACLE WEBLOGIC SERVER

As part of Oracle Fusion Middleware 11g, a family of standards based middleware products, WLS [7] is a Java application server

that provides, inter alia., a complete implementation of the JEE5 specification. JEE5 defines containers that manage user application components such as servlets (for web-based applications), Enterprise Java Beans (EJB), Message Driven Beans (MDB), application clients (outside the server container), and services that containers provide to user application components such as Java Database Connectivity (JDBC) for accessing SQL databases, Java Messaging Service (JMS) for reliable asynchronous point-to-point and publish-subscribe communication, and JTA for coordinating 2PC transactions. User applications typically access container services by looking up relevant API entry points in a directory implementing the Java Naming Directory Interface (JNDI) provided by the application server. Specifications and documentation about JEE5 are linked from the online reference [10].

In WLS, users register JMS and JDBC resources needed by their application components by means of corresponding deployment modules. JMS modules configure JMS client objects such as (potentially XA-enabled) connection factories, and message destinations (queues or topics). A JDBC module provides a definition of a SQL datasource, a collection of parameters, such as the name of a (potentially XA-capable) JDBC driver and the JDBC URL. JDBC datasources provide database access and database connection management. Each datasource defines a pool of database connections. User applications reserve a database connection from the datasource connection pool by looking up the datasource (*java.sql.DataSource*) on the JNDI context, and calling *getConnection()* on the datasource. When finished with the connection, the application should call *close()* on it as early as possible, which returns the database connection to the pool for further reuse. Using even an ideal XA JDBC driver is in general more expensive even in the single-resource 1PC transaction case: extra round-trip *XAResource* methods *start* and *end* have to be called regardless of the number of resources participating in the transaction.

WLS ships with integrated JMS provider (i.e., a JMS server) that can use a local filesystem or a JDBC datasource as an underlying message persistence mechanism.

The WLS JTA implementation consists of a distributed service that runs one TM instance per WLS server. A transaction may span multiple WLS servers (e.g., when an application invokes remote EJB's), in which case the TM's form a coordinator tree with the global coordinator being the root node. Each TM maintains persistent state for 2PC transactions in its transaction log for crash recovery purposes.

All WLS services on a server instance that require persistence including the JMS servers and the TM can be configured to use a single transactional Persistence Store that enables them to appear as a single resource in the 2PC. Persistent Store is either a File Store, a file-based implementation optimized for fast writes, or a JDBC Store utilizing a SQL database. Customers with high performance requirements use File Stores.

3. LLR DESIGN CONSIDERATIONS

In contrast to the Nested 2PC Optimization, we do not want to invoke prepare sequentially in a nested manner because it increases the overall latency to approximately (*n-1*)(*rtt*+*dst*), where *rtt* is a roundtrip time between two resources and *dst* is an average hard disk sync time. This part of the Nested 2PC Optimization saves the number of messages and was targeted three decades ago to expensive thin network links and frequent rollbacks, which we no longer experience today in the broadband age. However, we still need to single out one resource that we want to commit directly without calling *prepare*. The TM will call asynchronously *prepare* on *n-1* resources. If the TM has received *n-1 ACK*'s, it will invoke *commit* on the n^{th} resource (Last Resource) and the success of this call is used as a global decision about whether to call *commit* on *n-1* resources prepared during the voting phase as well.

The following points suggest choosing a JDBC datasource as an LLR:

- In a typical scenario, an application in WLS accesses a local JMS server, and remote database servers, and such a (sub)transaction is coordinated by the local TM. Passing 2PC messages to a local resource has a negligible cost of a local method invocation in Java because we use collocation. Therefore, a JDBC datasource connection is a perfect candidate to be selected as the Last Resource.
- In contrast to JMS messages, database data structures such as data records, primary and secondary indices are subjects to frequent updates. Committing a database transaction in one phase without waiting for the completion of the voting phase greatly reduces the duration of database locks. This potentially more than doubles the database transaction throughput. Therefore, even when the application accesses a remote JMS server (and the collocation argument no longer holds), the user will want to use a JDBC datasource as an LLR.
- When any component involved in a PA2PC transaction fails during the commit phase, the progress of this PA2PC instance (transaction) blocks until the failed component is restarted again. Reducing the number of components minimizes the risk of running components being blocked by failed ones. The LLR optimization enables the TM to eliminate the local Persistent Store resource by logging the global transaction outcome directly to the LLR. Hence, applications that do not involve local JEE services using the local Persistent Store other than JTA TM benefit even more from the LLR optimization.
- 2PC provides only Atomicity and Durability in the ACID contract; it does not provide global concurrency control (serialization, isolation) for global transactions [13]. Even worse, the order in which the changes made in different branches within the same global transaction become visible is not defined. LLR adds some intra-transaction determinism by defining the partial order: updates to the LLR branch are visible before updates to other branches. An application that makes an update to a JDBC datasource that is an LLR and sends a notification about this particular update to a JMS destination in the same transaction is guaranteed that the JDBC update is visible at the time when the notification is received.

The decision to use a JDBC resource as the last transaction outcome decider poses another challenge. Nested 2PC Optimization builds upon the fact that every resource in the chain is capable of being a coordinator, i.e., to persist the transaction outcome and continue the protocol execution after a crash. However, by design, there is no mechanism in a vanilla (XA) JDBC driver to remember transaction outcomes. The database is obliged by the ACID contract to preserve the updates of a committed transaction, but it will discard the transaction information from the log as we explained above for PA2PC.

```
update(XAResource xares, Transaction t,…)
1.  if xares is LLR accept only if t.llr is null
    and set t.llr = xares.
2.  if xares is not LLR and t.enlisted doesn't
    contain xares
        a.  t.enlisted.add(xares)
        b.  xares.start(t.xid)
3.  Perform updates via xares' connection

commit(Transaction t)
1.  for each xares in t.enlisted submit concur-
    rent tasks { xares.end(t.xid);
    xares.prepare(xid); }
2.  wait for prepare tasks to finish
3.  if t.llr is not null
        a.  t.llr.insert(t.xid)
        b.  t.llr.commit(), hence commit is test-
            able by looking up xid in the t.llr's
            table.
4.  if t.llr is null force-log commit record
    t.xid to local Persistent Store
5.  for each xares in t.enlisted submit concur-
    rent tasks { xares.commit(xid); }

On any error prior commit's 3.b the following is
called. Otherwise the LLR table is consulted.
error(Transaction t)
1.  for each xares in t.enlisted submit concur-
    rent tasks { xares.end(t.xid);
    xares.rollback(xid); }
2.  if t.llr is not null t.llr.rollback()
3.  throw exception to interrupt the caller

recover(TransactionManager tm):
1.  add commited xid's from local Persistent
    Store to tm.committedXidSet.
2.  add commited xid's from llr tables to
    tm.commitedXidSet.

when recreating JMS, and non-LLR XA JDBC re-
sources
recover(XAResource xares)
1.  unresolvedXidSet = xares.recover()
2.  for each xid in unresolvedXidSet
       a.  if xid is in tm.committedXidSet
           xares.commit(xid)
       b.  if xid is not in tm.committedXidSet
           xares.rollback(xid)}
```

Figure 2: Simplified pseudocode of LLR processing.

The solution to this problem is to maintain an extra LLR table in the schema connected by the Last Resource datasource. When the TM performs the LLR Optimization, instead of force-logging the transaction commit to the local Persistent Store it inserts an entry for the transaction id (XID) into the LLR table and only then the TM invokes commit on the Last Resource. This way we make sure that the commit of the Last Resource and logging the global transaction outcome is an atomic operation (hence, the name Logging Last Resource) (see *commit* in Figure 2). Therefore, the TM will be able to query the LLR table to determine the real transaction outcome even if the TM or the database fail once both operate normally again. Persisting of the transaction id as part of the transaction has also been used for recoverable ODBC sessions in a non-XA setting [1].

Once the TM committed the LLR transaction it issues asynchronously commit requests to the prepared XA resources. Receiving all outstanding commit request replies terminates the XA-2PC protocol. The WLS TM lazily garbage-collects in the LLR table log entries of terminated 2PC instances as part of subsequent transactions.

Now that we realize that we do not utilize the *XAResource* functionality to deal with the Last Resource, we can safely replace the XA JDBC driver in the LLR datasource definition by a faster non-XA version of the JDBC driver, which also eliminates expensive roundtrip calls to the *XAResource* methods *start* and *end*.

4. LLR PROCESSING DETAILS

On a datasource deployment or during the server boot, LLR datasources load or create a table on the database from which the datasource pools database connections. The table is created in the original datasource schema. If the database table cannot be created or loaded, then the server boot will fail.

Within a global transaction, the first connection obtained from an LLR datasource transparently reserves an internal JDBC connection on the transaction's coordinator. All subsequent transaction operations on any connections obtained from a same-named datasource on any server are routed to this same single internal JDBC connection.

Figure 2 sketches the LLR-modified processing of JTA transactions in WLS. During normal operation, using the LLR resource saves us at least one blocking call *start(xid)* when the LLR is being enlisted in the transaction (*update*), and one blocking call *end(xid)* when the LLR is being *committed.* Instead of calling *prepare* on LLR we call an insert statement to record the transaction xid in the LLR table, however, this call does not incur log forcing on the LLR. In contrast to Nested 2PC, all non-LLR resources execute 2PC concurrently. However, the commit of LLR is not started until the *prepare* phase of all non-LLR resources completes.

4.1 LLR Table

Each WLS instance maintains a LLR table on the database to which a JDBC LLR datasource pools database connections. These tables are used for storing 2PC commit records. If multiple LLR datasources are deployed on the same WLS instance and connect to the same database instance and database schema, they will also share the same LLR table. Unless configured explicitly, LLR table names are automatically generated as *WL_LLR_<serverName>*.

4.2 Failure and Recovery Processing for LLR

WLS TM processes transaction failures in the following way:

For 2PC errors that occur *before the LLR commit* is attempted, the TM immediately throws an exception. For 2PC errors that occur *during the LLR commit*: If the record is written, the TM commits the transaction; If the record is not written, the TM rolls back the transaction; If it is unknown whether the record is written, the TM throws an ambiguous commit failure exception and attempts to complete the transaction every five seconds until the transaction abandon timeout; If the transaction is still incomplete, the TM logs an abandoned transaction message.

During server boot, the TM will use the LLR resource to read the transaction 2PC record from the database and then use the recovered information to commit any subtransaction on any participating non-LLR XA resources.

If the JDBC connection in an LLR resource fails during a 2PC record insert, the TM rolls back the transaction. If the JDBC con-

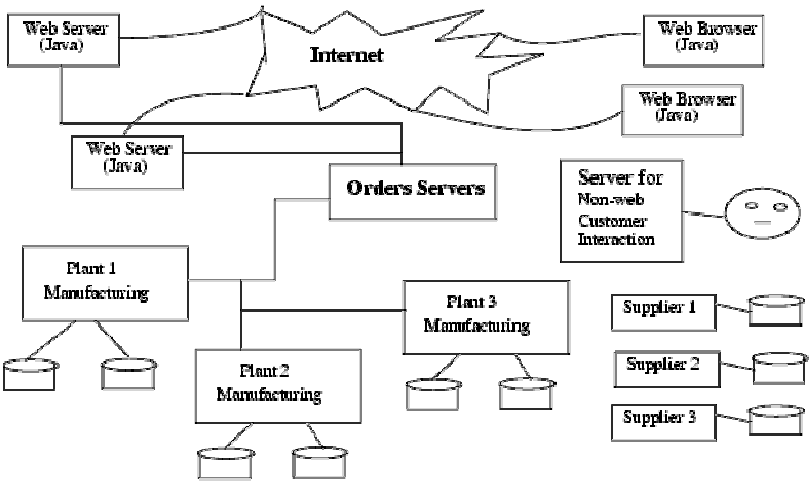

Figure 3: SPECjAppServer2004 world wide distributed business [9].

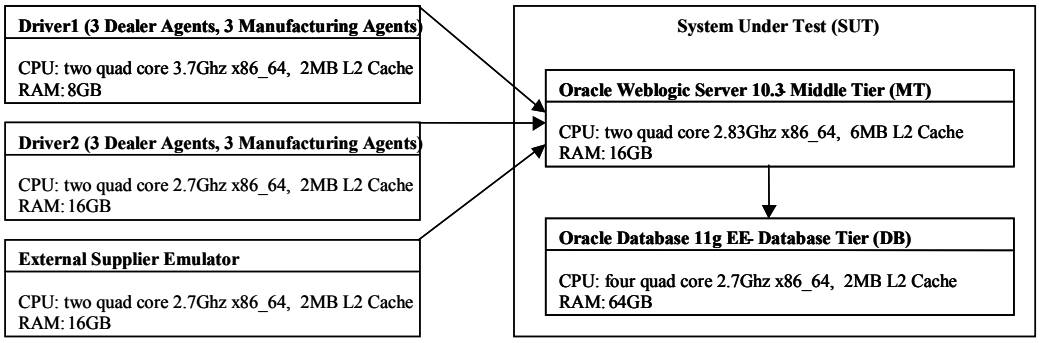

Figure 4: EAStress2004 setup

nection in an LLR resource fails during the commit of the LLR transaction, the result depends on whether the transaction is 1PC (i.e., when the LLR resource is the only participant) or 2PC:

For a 1PC transaction, the transaction will be committed, rolled back, or block waiting for the resolution of the LLR transaction. The outcome of the transaction is atomic and is determined by the underlying database alone. For a 2PC transaction, the outcome is determined based on the LLR table.

4.3 LLR Transaction Recovery

During server startup, the TM for each WLS must recover incomplete transactions coordinated by the server, including LLR transactions. Each server will attempt to read the transaction records from the LLR database tables for each LLR datasource. If the server cannot access the LLR database tables or if the recovery fails, the server will not start.

5. PERFORMANCE EVALUATION

LLR datasources have been used for the recent world record results achieved by Oracle Fusion Middleware in the SPECjAppServer2004 benchmark [6]. SPECjAppServer2004 is a benchmark of JEE-based application servers. It is an end-to-end application which exercises all major JEE technologies implemented by compliant application servers: The web container, including servlets and JSP's, the EJB container, EJB2.0 Container Managed Persistence, JMS and Message Driven Beans, Transaction Management, and JDBC [9]. Figure 3 depicts the way that these components relate to each other.

The SPECjAppServer2004 benchmark's research workload, coined EAStress2004, is used to evaluate the performance of LLR in this paper. SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004 / EAStress2004 results or findings in this paper have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result [9].

The five SPECjAppServer2004 domains are: Dealer, Manufacturing, Supplier, Customer, and Corporate. The Dealer domain encompasses the user interface components of a Web-based application used to access the services provided by the other domains (i.e., Manufacturing, Supplier, Customer and Corporate), which can be considered "business domains". There are producer-consumer relationships between domains in the company and to outside suppliers and customers as well.

Hardware used in our runs is depicted in Figure 4 where each box represents a computer node. Java components such as the benchmark drivers, the supplier emulator, and WLS are run by Oracle Jrockit JVM R27.6. The benchmark application accesses the database from the WLS using Oracle JDBC Driver 11g. XA datasources *oracle.jdbc.xa.client.OracleXADatasource* and non-

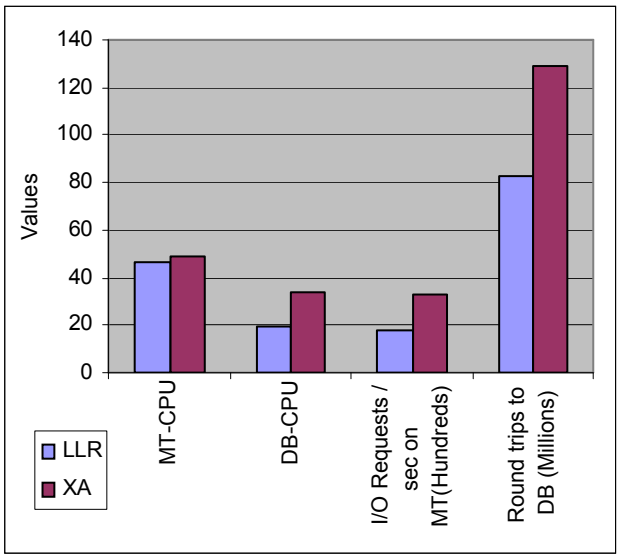

Figure 5: Resource utilization in EAStress2004.

XA datasources backed by *oracle.jdbc.OracleDriver* are used for XA and LLR runs, respectively.

The EAStress2004 workload is run with 10 minutes ramp up time, and 60 min steady state with Injection Rate (IR) 700 for both XA and LLR datasources. IR refers to the rate at which business transaction requests from the Dealer application in the Dealer Domain are injected into the system under test (SUT). Table 1 summarizes important results. Average transaction response times dramatically benefit from using the LLR Optimization.

In a run with an hour-long steady state, we observe that with XA datasources: DB CPU utilization increases by 80%, MT CPU goes up 6%. The round trips to the DB increase by around 50%. The I/O writes on the MT increase by 80%, with I/O service times remaining stable, but disk utilization increasing by 30%. Figure 5 depicts a resource utilization chart for the runs with and without LLR.

6. CONCLUSION

We introduced the LLR Optimization in WLS for XA-2PC involving JDBC datasource connections. LLR provides significant optimizations of and features beyond the original Nested 2PC in the context of JEE, which are unique to Oracle WLS: Failure during LLR resource recovery fails the boot process; LLR is cluster-ready: in a cluster, only the TC node hosts the LLR transaction's physical JDBC connection and requests submitted to identically named LLR datasources on other cluster members within the same transaction are rerouted through the coordinator; LLR log entries are automatically garbage collected. We are able to improve the JTA performance in WLS in a transparent fashion such that the user code does not require any changes. Solely the deployment descriptors or server-wide datasource definitions need to be redefined as LLR-enabled. Users need to make sure that their application deployments do not enlist more than one LLR datasource per global transaction. The LLR datasource connection is automatically selected for logging the global transaction outcome instead of the local Persistent Store. Successful commit of the LLR datasource commits the global transaction. Saving one phase on a remote datasource often more than doubles the transaction throughput. This feature is widely used by our customers in mission-critical applications with high performance requirements and has been validated by deploying it in recognized industry benchmarks [6].

Table 1: Summary of EAStess2004 v1.08 results for IR 700

	Response Time in seconds			
	Purchase	Manage	Browse	Manufacturing
XA	4.20	2.40	5.40	3.75
LLR	1.50	1.20	1.90	3.00
Delta	**2.8x**	**2x**	**2.8x**	**1.25x**

7. REFERENCES

[1] Barga, R., D. Lomet, T. Baby, and S. Agrawal: Persistent Client-Server Database Sessions. In Proceedings of 7th Int'l Conference on Extending Database Technology (EDBT), Konstanz, Germany, Mar 2000: 462-477

[2] Gray, J.: Notes on Data Base Operating Systems. In Advanced Course: Operating Systems, Springer, 1978: 393-481

[3] Lampson, D. and D. Lomet: A New Presumed Commit Optimization for Two Phase Commit. In Proceedings of the 19th Int'l Conference on Very Large Data Bases, Dublin, Ireland, Aug 1993, Morgan Kaufmann, San Francisco, CA, USA: 630-640

[4] Mohan, C. and B. Lindsay: Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. In Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles pf Distributed Computing, Montreal, Quebec, Canada, Aug 1983: 76-88

[5] Mohan, C., B. Lindsay, and R. Obermark: Transaction Management in the R* Distributed Database Management System. In ACM Transactions on Database Systems, 11(4), Dec 1986: 378-396

[6] Oracle Corp.: Oracle Benchmark Results, http://www.oracle.com/solutions/performance_scalability/benchmark_results.html

[7] Oracle Corp.: Oracle Fusion Middleware. http://www.oracle.com/products/middleware/index.html

[8] Oracle Corp.: Programming WebLogic JTA: Logging Last Resource Transaction Optimization. http://download.oracle.com/docs/cd/E12840_01/wls/docs103/jta/llr.html

[9] SPEC: SPECjAppServer2004, http://www.spec.org/jAppServer2004/

[10] Sun Microsystems: Java EE at a Glance. http://java.sun.com/javaee/.

[11] Sun Microsystems: Java Transaction API Specification 1.0.1, http://java.sun.com/javaee/technologies/jta/

[12] The Open Group: Distributed Transaction Processing: The XA Specification. X/Open Company Ltd., UK, 1991

[13] Weikum, G. and G. Vossen: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, San Francisco, CA, USA, 2001