



# Scalability Challenges in an InnoDB-based Replication Environment

**David Lutz**

Performance Engineer

Performance Applications Engineering

## Agenda

- Motivations
- Configuration and Initial Findings
- Analysis Tools and Techniques
- Code Inspection
- Prototype Improvements
- Results
- Futures

This presentation will highlight the tools and techniques that were used to develop a prototype patch to address two MySQL bugs that affect InnoDB/binlog scalability:

Bug#38501 "hold times on prepare\_commit\_mutex limit read/write scalability"

Bug#13669 "Group commit is broken in 5.0"

These issues came under focus as part of a broader scalability project being pursued by the MySQL and Performance Applications Engineering (PAE) organizations at Sun.

We will review the motivation for the scalability study, the process used to identify and investigate this particular issue, the development of a prototype patch, and the throughput improvements enabled by the patch. We will conclude with a brief look at other issues that need to be addressed to enable scalability within nodes in a MySQL replication environment.

## Motivations

- Multicore everywhere – Sun, AMD, Intel
- All nodes will soon be multi/many core
- Need to maximize scaling on each node
- Need to avoid negative scaling

This replication study was motivated by the proliferation of multi core systems from all current hardware vendors.

Even for traditional horizontal scale out environments, all nodes will soon be multi/many core and will need to scale to large numbers of hardware threads.

We would like to take advantage of available resources, and even more importantly avoid pathological behavior as resources are added (i.e. negative scaling).

Ideally we will see moderate to good scalability, with graceful degradation as we reach scalability limits.

## Testbed Configuration

- Sun T5240 – 2 sockets, 128 hardware threads, 32GB ram
- Solaris Nevada w/ sparse root zones
- Hardware RAID w/ write cache, ZFS file system
- MySQL 5.1.28 and 6.0.x w/ 10M row database (2GB)
- Sysbench read/write OLTP workload
  - > Software thread counts from 1 to 64

4

Page 4

The server used was a 128 thread CMT system, but could have been any system with a medium to large thread count.

A hardware write cache was used to enable the best possible write scaling and keep the focus on code/algorithm scaling.

MySQL 5.1.28 was used, and it should be noted that it contains performance fixes for other scalability bugs, including:

Bug#34409 LOCK\_plugin contention via ha\_release\_temporary\_latches/plugin\_foreach

Bug#38185 ha\_innbase::info can hold locks even when called with

HA\_STATUS\_NO\_LOCK

MySQL 6.0.5 with above patches added was also used in testing, and much of the initial work was actually performed there.

Sysbench was chosen as the workload because it is open source, simple to use, and seems to be popular in the MySQL community. The number of threads in the sysbench client were varied from 1 to 64, and the Transactions Per Second (TPS) were compared to assess scalability as thread counts were increased.

## MySQL Tuning

- Minimal my.cnf including:
  - > query\_cache\_size = 0
  - > table\_open\_cache=2048
  - > datadir = RAID w/ write cache
  - > innodb\_buffer\_pool\_size = 4096M
  - > innodb\_log\_group\_home\_dir = dedicated FS w/ write cache
  - > innodb\_flush\_log\_at\_trx\_commit=1
  - > innodb\_thread\_concurrency = 0
  - > sync\_binlog=0

5

Page 5

Query cache is disabled due to scalability issues with this workload.

table\_open\_cache size is increased to reduce LOCK\_open lock contention.

InnoDB buffer pool is made large to keep most reads in memory.

InnoDB logs on a separate FS and array from data to provide the best latency.

Flush logs for each commit in InnoDB.

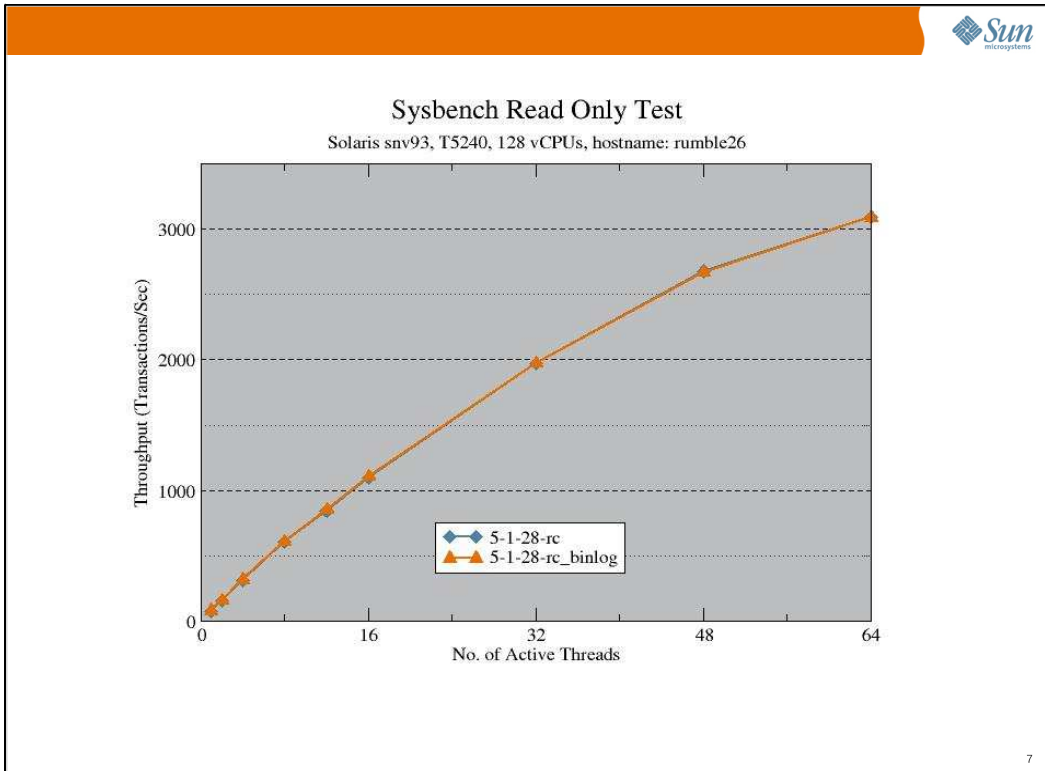
Allow unlimited InnoDB thread concurrency.

Note that **we have not set sync\_binlog=1**, which would tell MySQL to flush the binlog file with each InnoDB commit. This would be desirable, but is a performance drain that is not addressed by the current project.

There were a few other settings, but they aren't thought to be specific to this study.

# Initial Findings

See the next few slides...

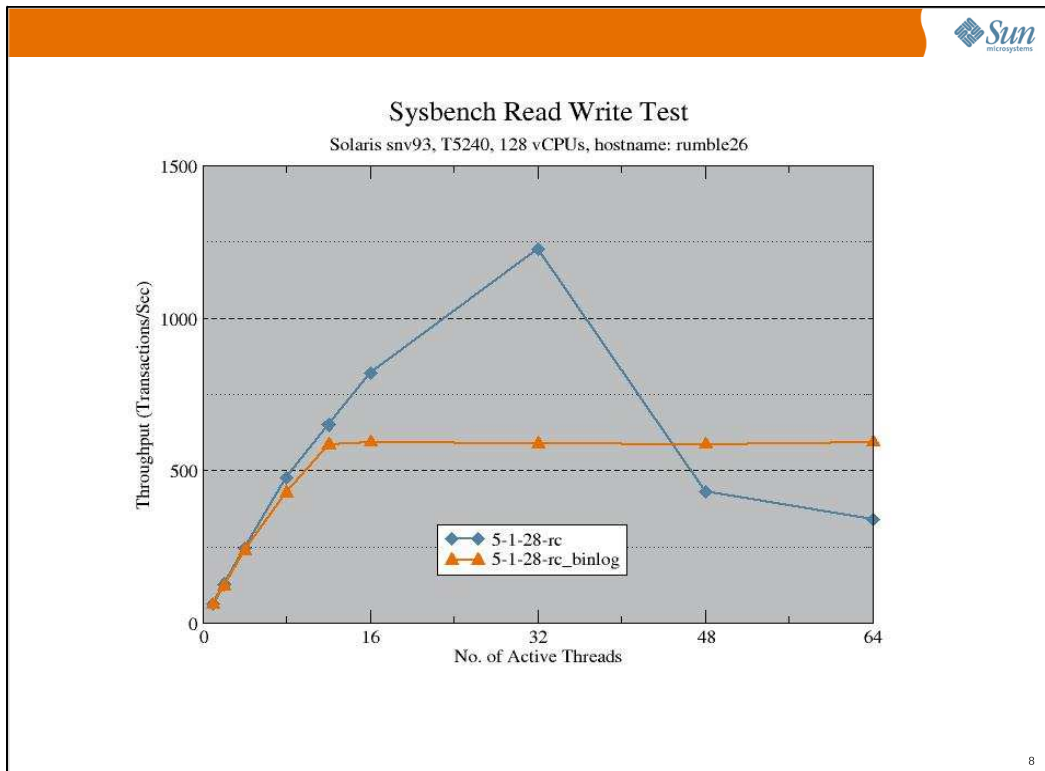


For read-only sysbench, shown above, we see:

Moderate read-only scalability, with little or no impact to scaling when binlog is enabled.

In both configurations, scaling is roughly linear to 4 threads, then begins to decrease. At 32 threads, scaling is roughly 78% of linear and at 64 threads it is roughly 61%.

Note that we begin to see negative scaling beyond 64 threads, although that data is not shown in the above graph.



For read-write sysbench **without** binlog logging (blue line), we see:

modest scaling to 32 threads, with negative scaling at 48 threads and higher. At 32 threads, scaling is roughly 61% of linear.

**Note that 6.0.x contains additional scalability improvements and shows continued scaling to 48 threads without binlog enabled.**

**With** binlog enabled (orange line), we see a substantial scaling impact.

Scaling is essentially flat at 12 threads and higher. Scaling with binlog logging is 30% of linear at 32 threads, translating to a peak cost for binlog logging of 53% of throughput at 32 threads.

The biggest concern here initially is the high cost in throughput when binlog logging is enabled, and this became the focus for a new patch.

# Analysis Tools and Techniques

See next slides...

## Finding Hot Locks

- Highly contended (Hot) locks limit scalability
- Two main causes
  - > High demand resource
  - > Long hold time on a lock
- Solaris plockstat (dtrace consumer)
  - > Contention: plockstat -C
  - > Hold times: plockstat -H
  - > Both: plockstat -A
- SunStudio Performance Analyzer

For serial performance improvements, we typically profile the code looking for functions/methods that consume a lot of time. For parallel scalability, we typically look for hot locks as a limit on scalability.

A high demand resource may result in a hot lock even if the time spent by each consumer of the resource is very small. This must be addressed by breaking up the resource, switching to a lock free algorithm, etc.

Long hold time on a lock may be due to taking the lock before it is needed, holding it after it is no longer needed, going to sleep while holding the lock (performing I/O, etc), or using a single lock to protect a large data structure or section of code. The fix depends on the situation, but in general we need to reduce the size of the “critical region” protected by the lock.

Holding a lock while performing I/O should definitely be avoided.

The Solaris plockstat utility can be used to find hot locks in user programs.

SunStudio Performance Analyzer can also collect and report on synchronization wait events in Solaris.

## Plockstat | c++filt

```
plockstat -C -n 10 -e 5 -x bufsize=10k -x aggsz=2m -p 3491 | c++filt  
Mutex block
```

Count	nsec	Lock	Caller
-----			
---			
2038	54747571	mysqld`\$XAJzoBKpX4GIETk. <b>prepare_commit_mutex</b>	mysqld`int <b>innobase_xa_prepare</b> (handlerton*, THD*, bool)+0x4c
282	447655	mysqld`mysql_bin_log+0x8 MYSQL_BIN_LOG::write(Log_event*)+0xa4	mysqld`bool
83	471277	mysqld`mysql_bin_log+0x8 MYSQL_BIN_LOG::write(Log_event*)+0xa4	mysqld`bool
...			

11

Page 11

This plockstat command was run at varying thread counts. The current example is from mysql 6.0.5-alpha-pb87 with 32 sysbench threads, and has been truncated due to space limits.

This utility often makes it extremely easy, and sometimes quite obvious, to identify the root cause for a scalability issue in a multi-threaded program.

The options used above include:

- n 10 = top 10 locks

- e 5 = sample for 5 seconds

- x bufsize=10k -x aggsz=2m = increase size of buffers in dtrace due to large number of events

- p 3491 = monitor process 3491, which was mysqld

Note that the output includes C++ mangled names. c++filt is part of SunStudio and converts mangled c++ names back to original code.

The above shows that the hottest lock is prepare\_commit\_mutex and the hottest caller is innobase\_xa\_prepare().

Searching through the code for this mutex and function call showed that the mutex was acquired in innobase\_xa\_prepare() and released in innobase\_commit(). This code is located in storage/innobase/handler/ha\_innodb.cc.

A Google search showed that the code was added during the 4.1 to 5.0 upgrade, as part of XA transaction support. Incidentally, this specific code section is a very tiny part of a major code upgrade.

## Pstack

```
$ pstack 3491 | c++filt
...
----- lwp# 150 / thread# 150 -----
ffffffff7ddcbf64 lwp_park (0, 0, 0)
00000001003e794c int innobase_xa_prepare(handlerton*,THD*,bool) ...
0000000100336210 int ha_commit_trans(THD*,bool) ...
0000000100255714 int end_trans(THD*,enum_mysql_completiontype) ...
000000010025b518 int mysql_execute_command(THD*) ...
00000001002e0b8c bool Prepared_statement::execute(String*,bool) ...
00000001002dfa0c void mysql_stmt_execute(THD*,char*,unsigned) ...
...
```

12

Page 12

The Solaris pstack command displays a stack trace for every thread in the given process. The stack traces can show you who is calling a given function.

The stacks contain c++ mangled names, so pipe through the SunStudio c++filt utility to convert names back to original code.

This output is from mysql 6.0.5-alpha-pb87, and has been truncated to save space, removing arguments from each stack entry and truncating the stack. We also only show a single interesting stack; the output originally include a stack for every thread.

When looking for a function that is being held up by a highly contended lock, it is likely that you will find that caller in a stack dump after a few tries with pstack. The call to lwp\_park() at the top of the stack shows that this thread is stalled, probably waiting for that lock.

A more certain way to display the stack traces is to use the dtrace pid provider and the ustack() function to dump a stack trace on entry to the given function, as shown in the next slide.

## Dtrace ustack()

```
#!/usr/sbin/dtrace -qs

pid$target:::__1cTinnobase_xa_prepare6FpnKhandlerton_pnDTHD_b_i_:entry
{ @hash[ustack(50)] = count(); }

END

{ trunc(@hash, 5); }
```

13

Page 13

The dtrace script show above can be run with:

```
./script_name.d -p NNN
```

where NNN is the pid of the mysqld process.

After running for a period of time, say 20 seconds, ^C to end the script and you will see the top 5 most frequent stacks that end with `prepare_commit_mutex()`.

Note that you must use the c++ mangled name for the function, and that the output will include c++ mangled names.

You can save the output and pipe it through SunStudio c++filt to convert back to original code.

If you don't know the c++ mangled name for a function you want to track, you can search for it in the output from `/usr/ccs/bin/nm libexec/mysqld`

## Dtrace call tracing

```
#!/usr/sbin/dtrace -qs
#pragma D option flowindent
pid$target::__1cTinnobase_xa_prepare6FpnKhandlerton_pnDTHD_b_i_:entry
{ self->in = 1; }
pid$target:::entry,
pid$target:::return
/self->in/
{ printf("\n"); }
pid$target::__1cPinnobase_commit6FpnKhandlerton_pnDTHD_b_i_:return
/self->in/
{ self->in = 0;
  exit(0);
}
```

14

Page 14

We knew from plockstat that the prepare\_commit\_mutex was acquired in innobase\_xa\_prepare().

Searching source code we determined that the lock was released in innobase\_commit(). We wanted to see what happens between these two calls.

The dtrace script shown above starts tracing on entry to innobase\_xa\_prepare() and stops tracing on return from innobase\_commit().

Both functions are named using their c++ mangled names.

This particular script is VERY expensive to run because it instruments a large number of function entry and exit points, but it gives a nice indented listing of the calls. This is very helpful in understanding what happens between the execution of these two functions.

## Dtrace call tracing output

```
$ pfexec ./innobase_xa_prepare_trace.d -p 3491

CPU FUNCTION
48  -> __1cTinnobase_xa_prepare6FpnKhandlerton_pnDTHD_b_i_
48  -> __1cQcheck_trx_exists6FpnDTHD__pnKtrx_struct__
48    -> thd_ha_data
48    <- thd_ha_data
...
48    -> srv_active_wake_master_thread
48    <- srv_active_wake_master_thread
48  | __1cPinnobase_commit6FpnKhandlerton_pnDTHD_b_i_:return
48  <- __1cPinnobase_commit6FpnKhandlerton_pnDTHD_b_i_
```

15

Page 15

The output above shows the results of running the dtrace script from the previous page. It has been truncated for space.

This listing is for a single thread, showing each function/method as it is called. Children are indented from their parent. Entry to a function is indicated by '->' while '<-' is used to indicate return. The actual output might show data from more than one thread.

Note that the output includes c++ mangled names. This can be piped through the SunStudio c++filt utility, but this will change the indenting so it is good to look at the output both ways.

## Pseudocode – 5.0, 5.1, 6.0

```
mysql_execute_command() -> end_trans() -> ha_commit_trans()
    innobase_xa_prepare()
        acquire prepare_commit_mutex lock
        trx_prepare_for_mysql() -> write undo log
        MYSQL_BIN_LOG::log_xid()
        acquire binlog lock
        write binlog
        release binlog lock
    ha_commit_one_phase() -> innobase_commit()
        innobase_commit_low() -> trx_commit_for_mysql()
        write redo log
        release prepare_commit_mutex lock
```

16

Page 16

Code inspection revealed the basic steps outlined above, which have been greatly simplified.

The main components are found in `sql/sql_parse.cc`, `sql/handler.cc`, and `storage/innobase/handler/ha_innodb.cc`

Key points:

- we acquire the `prepare_commit_lock` very early in `innobase_xa_prepare()`
- we then write the InnoDB undo log, the MySQL binlog, and finally the InnoDB redo log before releasing the lock
- this is done to maintain ordering between InnoDB and MySQL logs.
- It is very strongly encouraged to flush the InnoDB log after each write to ensure transactions are not lost if there is a system crash
- It is encourage, but less strongly, to flush the binlog file after each write
- this completely serializes all InnoDB write transaction commits for the time it takes to write and flush these files.

## Pseudocode – 4.1

```
mysql_execute_command() -> ha_commit_trans() -> MYSQL_LOG::write()  
  
    acquire binlog lock  
  
    write binlog  
  
    ha_report_binlog_offset_and_commit()  
  
        write InnoDB log without flush  
  
    release binlog lock  
  
    ha_commit_complete()  
  
        flush InnoDB logs
```

17

Page 17

A good first step in trouble shooting is to find an instance that works and see what changed.

The `prepare_commit_mutex` lock was introduced as part of XA transaction support that was added in the 4.1 to 5.0 upgrade.

We examined the 4.1.22 code to see how things worked there. The main components are found in `sql/sql_parse.cc`, `sql/handler.cc`, `sql/log.cc`, and `sql/ha_innodb.cc`.

The big change is that the binlog lock is used to protect the binlog file and to protect the storage engine log.

The binlog file is written and possibly flushed while holding the lock, but the storage engine log is only written without a flush while holding the lock.

After releasing the binlog lock, the storage engine log is flushed.

This means that other transactions are able to continue without waiting for the storage engine flush to complete, substantially reducing lock hold time and the serialization of transactions.

## Prototype Improvement

- See MySQL Bug#38501 and Bug#13669
- Acquire prepare\_commit\_mutex later
  - > Near the end of innobase\_xa\_prepare()
  - > After undo log is written
- Release lock before redo log flush
  - > Turn off flush
  - > Write log
  - > Release lock
  - > Perform flush

This issue was filed as Bug#38501 "hold times on prepare\_commit\_mutex limit read/write scalability".

It was later determined to be the same issue as Bug#13669 "Group commit is broken in 5.0"

Examining the 5.0 and above code, it was determined that the basic functionality was available with some fairly minor code changes.

The biggest change is that we write to the InnoDB redo log without flushing to disk, then release the prepare\_commit\_mutex lock, then flush the redo log.

This allows other transactions to continue as soon as the redo log write is written to the file system buffer, rather than having to wait for the disk flush to complete.

This results in a substantial reduction in hold time on the lock, reducing serialization of write transactions.

This also restores group commit functionality, since multiple threads may write to redo log buffer before the commit takes place.

## Prototype Pseudocode

```
mysql_execute_command() -> end_trans() -> ha_commit_trans()
    innobase_xa_prepare()
        trx_prepare_for_mysql() -> write undo log
            acquire prepare_commit_mutex lock
        MYSQL_BIN_LOG::log_xid()
            acquire binlog lock
            write binlog
            release binlog lock
        ha_commit_one_phase() -> innobase_commit()
            innobase_commit_low() -> trx_commit_for_mysql()
                write redo log without flush
            release prepare_commit_mutex lock
            flush redo log
```

19

Page 19

Note that we now acquire the `prepare_commit_mutex` lock after writing the undo log, then write the binlog file while still holding the lock. If `sync_binlog=0`, this write is to a file system buffer only, with no synchronous flush to disk.

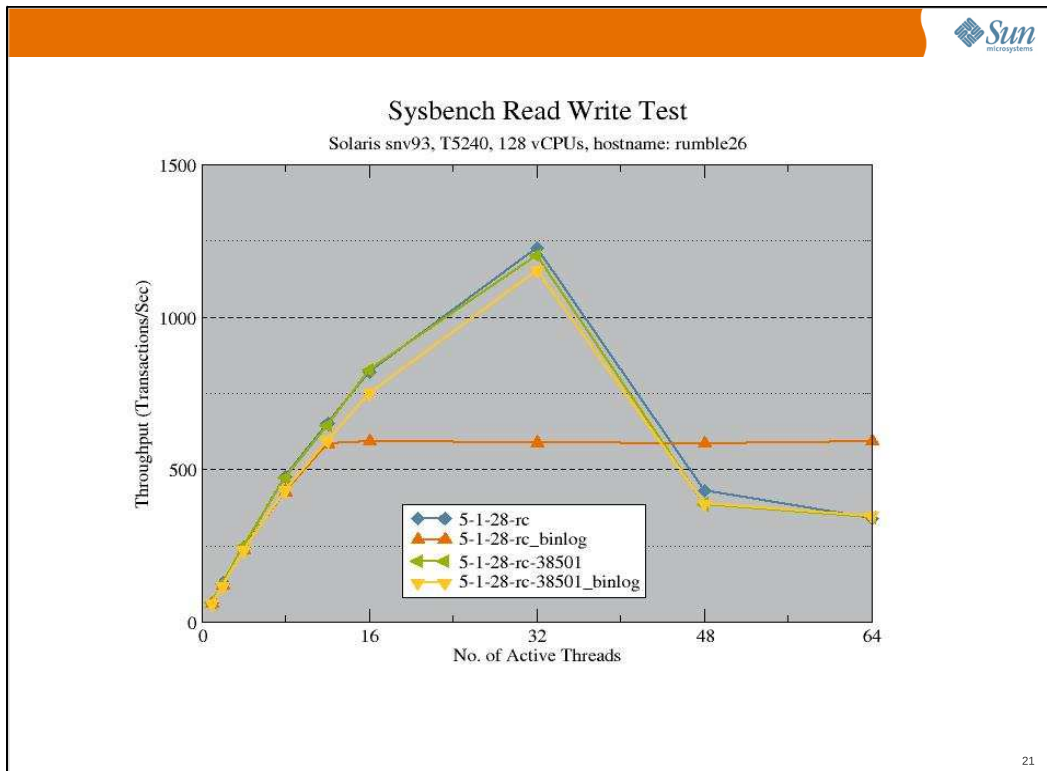
To perform the redo log write without a flush, we set `trx->flush_log_later = true` before calling `innobase_commit_low()`. We then restore the original `trx->flush_log_later` value and release the `prepare_commit_mutex` lock.

After releasing the lock, we call `trx_commit_complete_for_mysql()` directly, which flushes the redo log to disk if `trx->flush_log_later` is false.

This code is found in `storage/innobase/handler/ha_innodb.cc` and `storage/innobase/trx/trx0trx.c`

# Results

See next slides...

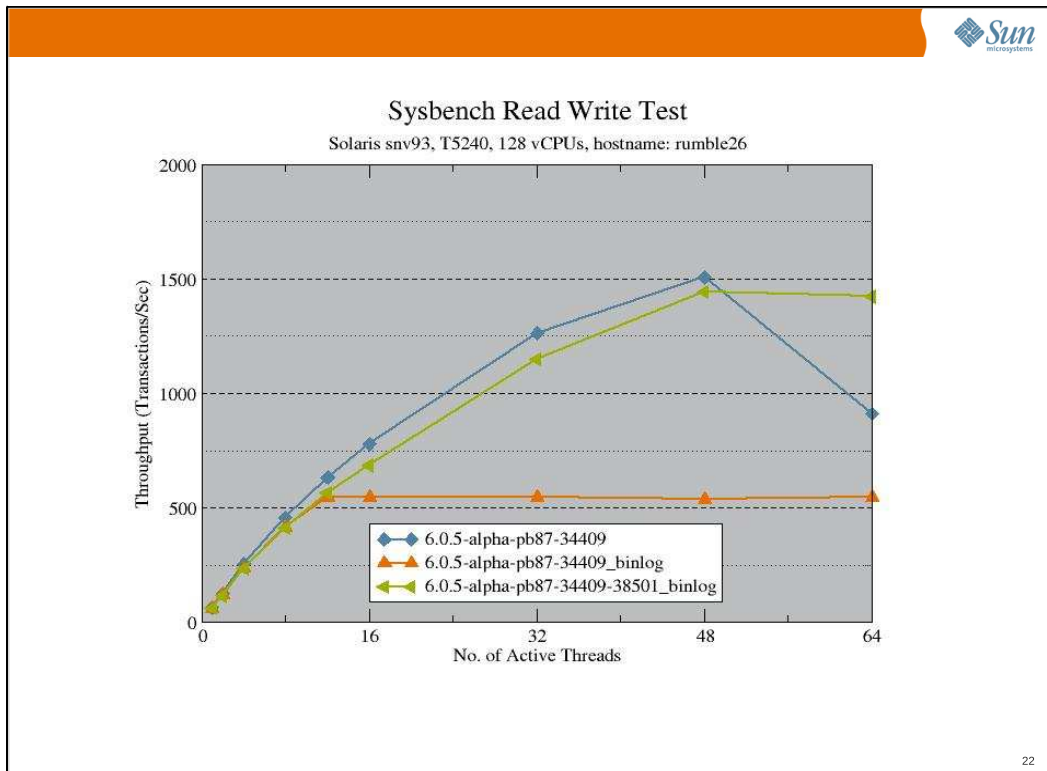


The chart shown above compares 5-1-28-rc with and without binlog logging enabled (the blue and orange lines, respectively) to 5-1-28-rc with the prototype patch for bug 38501, also with and without binlog logging (the green and yellow lines, respectively).

This shows a **1.95X improvement** in throughput for the patched version with binlog enabled.

This also shows that with the prototype patch, throughput with binlog enabled is only 6% below throughput without binlog enabled, compared to 53% without the patch.

Not shown here is the reduction in disk writes per transaction from ~2 to ~0.5 due to coalescing of concurrent commits.



The chart above compares 6.0.5-alpha-pb87 with a prototype patch for bug 34409, with and without binlog enabled (the blue and orange lines, respectively), to the same version with the prototype patch added for 38501 (the green line).

As compared with 5.1.28-rc, this shows that the fix for 38501 continues to scale along with the underlying 6.0.5 code, out to 48 threads.

With this code base, we see a **2.5X improvement** in throughput for patch 38501 with binlog enabled.

It has not yet been determined why the version with the prototype patch and with binlog enabled showed better throughput at 64 threads than the version without the prototype patch and without binlog. This is a positive result for the prototype patch, but isn't predicted by the other data.

## Futures

- Get verification/approval from Innobase for the changes just described
- Reduce slave side serialization
- Reduce serialization w/ sync\_binlog=1
- Reduce contention on
  - > LOCK\_log binlog lock
  - > InnoDB Sync array
- General MySQL scalability improvements

The first issue that needs to be addressed is to get verification and approval by Innobase for the prototype patch described here. Some of the other areas that need to be targeted for future performance improvements are noted above.

Slave side serialization is one of the biggest issues, since updates to the slaves are essentially 100% serialized at this time. This means that there can be a substantial lag in data propagation to the slaves, and this lag will become greater as we increase write scalability on the master. The MySQL replication team is already working on solutions for this issue.

Improvements for behavior with sync\_binlog=1 are needed because synchronous binlog file updates serialize transactions even with the prototype patch discussed in this presentation. What is needed is a way to allow synchronous writes to multiple files within the life of a given transaction, without serializing those synchronous writes and while allowing concurrent synchronous writes from other transactions. This might be achieved by sending the writes to worker thread queues, then notifying the waiting threads when the I/O is complete.

Reductions in contention on LOCK\_log and InnoDB sync array locks are also needed, since these show up as the hottest locks in some workloads after applying the patch for bug 38501. LOCK\_log contention might be addressed by the same solution mentioned in the previous paragraph. Reductions in sync\_array lock contention are being considered separately.

General MySQL scalability is a major focus of the Sun Performance Technologies group, and research is taking place on a number of platforms and workloads. We should see improvements in many areas in the coming months.



Questions?

David Lutz  
[David.Lutz@Sun.COM](mailto:David.Lutz@Sun.COM)