

BTM-2.2

The changes to BTM-2.2 are the result of extensive profiling using JProfiler 7.0, including CPU-time analysis as well as lock-contention analysis. For these analyses, the profiler was restricted as much as possible to just the Bitronix components.

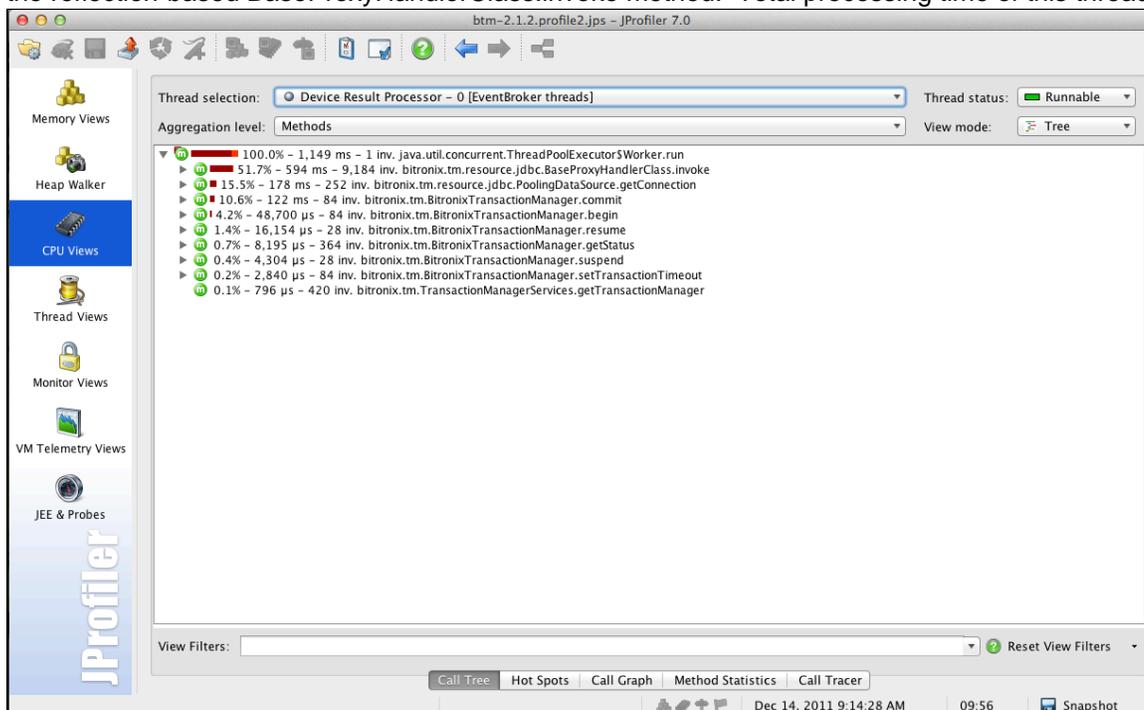
This page provides a comparison of the BTM-2.1.2 release vs. the BTM-2.2 branch in a real world load test. These profiling runs include only one of the workloads in my company's product, but is the most important one for customers. Having said that, I would expect the results of this analysis to be representative of other high-concurrency workloads.

CPU

First, let's look at CPU usage. BTM-2.2 contains a fair number of concurrency changes, but also a move away from the BTM-2.1.2 "proxy" wrapper around JDBC objects. The proxy wrapper was designed to allow bitronix to support JDBC3 and JDBC4 without a compile-time dependency on JDBC4. Unfortunately, this has sacrificed runtime performance in favor of build-time simplicity.

BTM 2.1.2

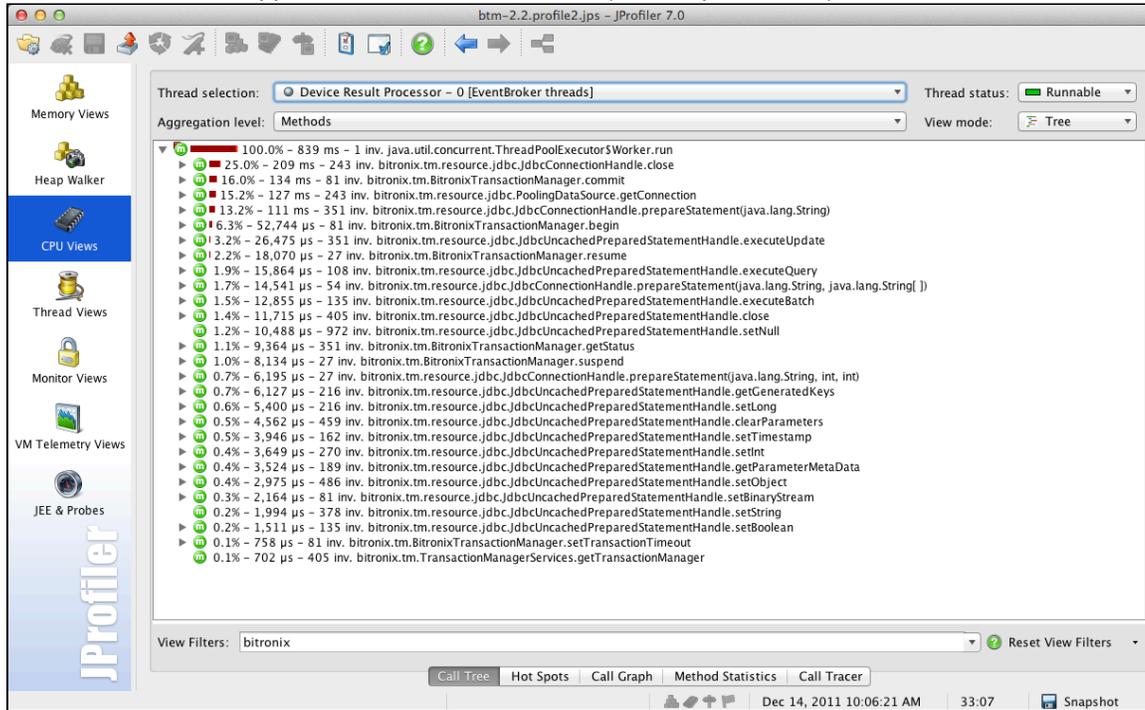
As can be seen here, looking at one of the processing threads in the workload, 594ms were spent by this thread in the reflection-based `BaseProxyHandlerClass.invoke` method. Total processing time of this thread was 1149ms.



BTM-2.2

BTM-2.2 replaces the reflection-based approach with concrete JDBC3 and JDBC4 classes that wrap and delegate. The cost is increased build complexity, but improved runtime performance. As can be seen here, looking at the

same processing thread as above in the workload, the reflection invocation overhead is gone, and the total CPU time of the thread dropped from 1149ms to 839ms (27% improvement).



You might notice in this call tree, when compared to the BTM-2.1.2 call tree, a lot of new methods appear: `JdbcConnectionHandle.close()`, various `PreparedStatement` methods, etc. These were previously contained "within" the proxy invocation, but now appear as direct calls. Subtracting the two, we find that the proxy overhead was approximately 300ms for this thread. 300ms out of the original 1149ms is a rather large percentage.

Monitors (lock contention)

While CPU time is important, and directly impacts the overall "load" on a server, another performance killer is lock contention. In lock contention, very little CPU time is expended, but threads are blocked from performing useful work. This extends the overall runtime of a given workload. One area that immediately showed up in profiling bitronix was the XAPool resource pool. One or two threads contending for a resource may experience very little lock contention, but as the number of processing threads increases into the 10s or 100s, lock contention can become the primary bottleneck in a system.

BTM-2.1.2

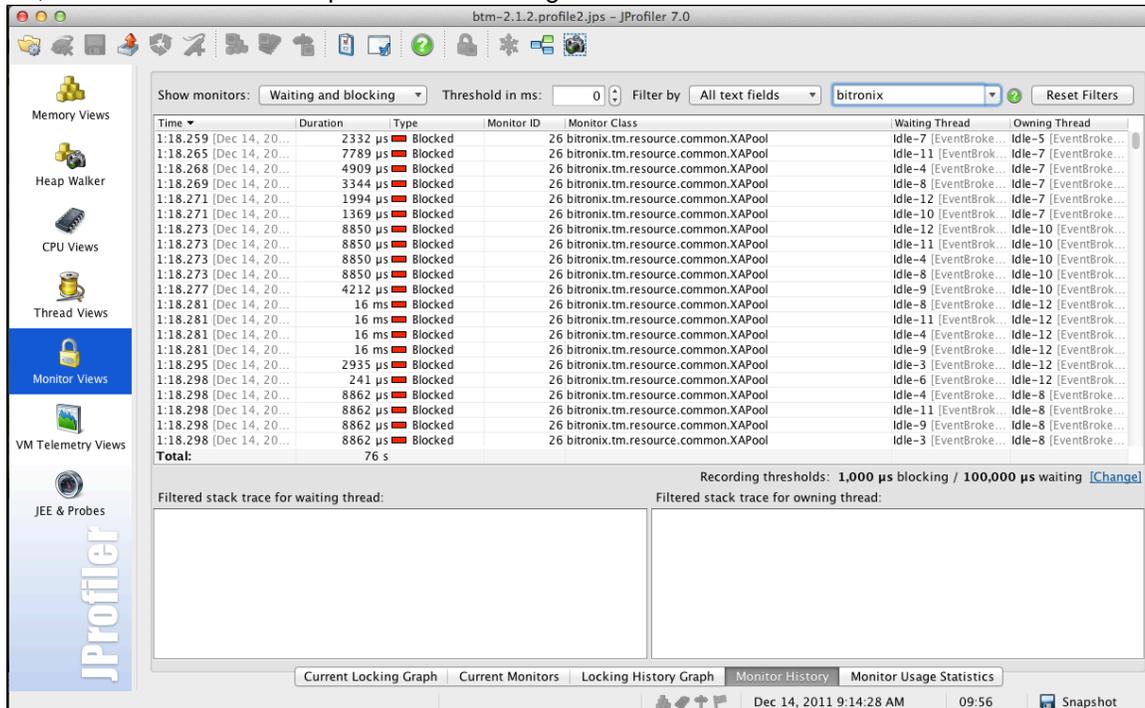
In BTM-2.1.2, the XAPool is largely synchronized by a high-level lock (synchronized on the XAPool). In addition, *while holding this lock* various actions are performed. These include:

- iteration over the pool is performed to look for "NOT_ACCESSIBLE" resources (`getNotAccessible()`)
- iteration over the pool to count "IN_POOL" resources (`inPoolSize()`)
- iteration over `XAStatefulHolder` objects to find the `XAResourceHolder` (`findXAResourceHolder()`)
- iteration over the `XAStatefulHolder` objects to shrink the pool (`shrink()`)
- iteration over the `XAStatefulHolder` objects to find an "IN_POOL" resource (`getInPool()`)
- growing the pool by adding new connections (`grow()`)

Performing these actions while holding the lock increase lock contention substantially in a high-concurrency environment.

Here is a screenshot from JProfiler's Monitor Views showing the number of times the XAPool lock was contended

for, and total time threads spent blocked during the load test:



Looking at the total time, at the bottom of the Duration column, you can see that a total of ~76 seconds was spent blocked on XAPool during the workload. While each thread blocked for on average only 8ms, some blocked for as long as 390ms.

As you can see from the size of the scrollbar on the right-side of the view, there were a lot of lock contentions. In fact, there were 26269 times that threads were blocked interacting with XAPool.

I encourage you to download the attached zip file, and open the BTM-2.1.2-monitor-history.html file. Its size (13.4MB) gives an indication of how many lock contentions occurred.

BTM-2.2

In BTM-2.2, the XAPool was refactored, along with small changes to related classes. This is a summary of the changes:

- the high-level lock on XAPool was removed from the primary code path.
- the single pool collection was split into three pools: availablePool (IN_POOL), inaccessiblePool (NOT_ACCESSIBLE), and accessiblePool (ACCESSIBLE)
- a ReentrantReadWriteLock was introduced to ensure that the movement of a connection from one pool to another is atomic
 - the read-side of the lock allows multiple threads ("readers") to iterate concurrently. They are guaranteed that a modification will not occur while they are iterating.
 - the write-side of the lock protects the instant in time when a connection changes state (for example from NOT_ACCESSIBLE to IN_POOL). When a connection is moving, "readers" are not allowed to enter the lock (but the duration of the write-lock is measured in microseconds). Conversely, when readers are iterating, a "writer" is blocked from moving a connection until they exit.

By splitting the pools, and using new collection-types the following advantages are gained (compare one-by-one to the BTM-2.1.2 bullets above):

- iteration over the inaccessiblePool is still performed (getNotAccessible()), but...
 - any number of threads are allowed to iterate, increasing concurrently substantially

- *only* the XAStatefulHolders in the inaccessiblePool are iterated. BTM-2.1.2 had to iterate *all* XAStatefulHolders and ignore IN_POOL and ACCESSIBLE entries.
- the availablePool is a BlockingQueue. this means inPoolSize(), rather than iterating over the entire single pool to count "IN_POOL" resources (and ignore NOT_ACCESSIBLE and ACCESSIBLE), simply calls availablePool.size().
- in BTM-2.2, XAPool.findXAResourceHolder() was moved.
 - in BTM-2.1.2, XAPool.findXAResourceHolder(XAResource xaResource) performed a "double-iteration". first, it iterated over all XAStatefulHolders in the pool, then for each holder, it iterated over all XAResourceHolders until it found the holder for the requested resource. this was performed within the lock, because iteration was over the single pool collection.
 - in BTM-2.2, findXAResourceHolder(XAResource xaResource) was moved to the XAResourceProducer interface. implementers of the XAResourceProducer interface (for example, PoolingDataSource) now use a private Map<XAResource, XAResourceHolder) which allows the find to be performed without iteration (and without a lock).
- the shrink() method now runs concurrent with other pool operations
 - in BTM-2.1.2, shrink() had to iterate over the entire pool, ignoring NOT_ACCESSIBLE and ACCESSIBLE XAStatefulHolders. close() also happened within the context of the lock.
 - in BTM-2.2, shrink() now only needs to iterate the accessiblePool, rather than iterating over and ignoring other XAStatefulHolders. it is also done without blocking any readers.
- the getInPool() method can now simply perform a poll() from the availablePool collection (BlockingQueue)
 - in BTM-2.1.2, a side-effect of getInPool() was that grow() might be called. this meant that the relatively slow operation of creating new connections occurred within the lock, preventing any other threads from obtaining connections during the process.
 - in BTM-2.1.2, getInPool() had to potentially iterate over all XAStatefulHolders and ignore NOT_ACCESSIBLE and ACCESSIBLE connections
 - in BTM-2.2, grow() might still be called, but there is no lock that is held, so other threads are not prevented from obtaining connections (for example NOT_ACCESSIBLE or thread-shared connections).
 - using the BlockingQueue.poll() method that accepts a timeout (when obtaining a resource from the pool) also allows integration of the pool timeout feature "for free"
- the grow() method now occurs outside of the context of any high-level lock. new XAStatefulHolders are created, and their addition to the availablePool are atomic by nature of the concurrent collection.

I include these probably obvious observations about the BTM-2.1.2 XAPool:

- Because of the synchronized nature of XAPool.getConnectionHandle(), only one thread can obtain a connection at a time.
- Within the context of that lock, connections are tested to see if they are still alive. In my configuration, is penalty is fairly "low" because I use JDBC4 connections. But in the case of users who actually run a test query, all threads wanting a connection must wait on the lock until the database actually responds to the test query of the single acquiring thread.

While all of these changes sound extensive, the XAPool code is now 565 lines (ignoring added JavaDoc) compared to 524 lines in BTM-2.1.2. Additionally, while the improvements in concurrency brought by these changes might seem abstract or academic, they are anything but that. In fact, i don't hesitate to say the results are astounding.

For the exact same workload, here is a screenshot from JProfiler's Monitor Views showing the number of times a lock within XAPool was contended for, and total time threads spent blocked during the load test:

btm-2.2.profile2.jps - JProfiler 7.0

Show monitors: **Waiting and blocking** Threshold in ms: Filter by: **All text fields**

Time	Duration	Type	Monitor ID	Monitor Class	Waiting Thread	Owning Thread
1:23.430	Dec 14, 20...	11 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-9 [EventBroker th...	Idle-11 [EventBroker...
1:24.171	Dec 14, 20...	13 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-7 [EventBroker th...	Idle-6 [EventBroker t...
1:24.171	Dec 14, 20...	13 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-9 [EventBroker th...	Idle-6 [EventBroker t...
1:24.184	Dec 14, 20...	50 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-7 [EventBroker th...	Idle-9 [EventBroker t...
1:24.185	Dec 14, 20...	49 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-5 [EventBroker th...	Idle-9 [EventBroker t...
1:24.208	Dec 14, 20...	26 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-3 [EventBroker th...	Idle-9 [EventBroker t...
1:24.235	Dec 14, 20...	22 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-3 [EventBroker th...	Idle-7 [EventBroker t...
1:24.235	Dec 14, 20...	22 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-5 [EventBroker th...	Idle-7 [EventBroker t...
1:24.258	Dec 14, 20...	17 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-5 [EventBroker th...	Idle-3 [EventBroker t...
1:25.217	Dec 14, 20...	2289 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-8 [EventBroker th...	Device Result Process...
1:26.000	Dec 14, 20...	16 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Device Result Process...	Device Result Process...
1:26.002	Dec 14, 20...	14 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-7 [EventBroker th...	Device Result Process...
1:26.011	Dec 14, 20...	5442 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-4 [EventBroker th...	Device Result Process...
1:26.017	Dec 14, 20...	6656 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-7 [EventBroker th...	Idle-4 [EventBroker t...
1:26.017	Dec 14, 20...	6656 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Device Result Process...	Idle-4 [EventBroker t...
1:26.023	Dec 14, 20...	8222 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Device Result Process...	Idle-7 [EventBroker t...
1:26.029	Dec 14, 20...	2493 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-11 [EventBroker...	Device Result Process...
1:26.031	Dec 14, 20...	11 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-11 [EventBroker...	Device Result Process...
1:26.047	Dec 14, 20...	9798 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Device Result Process...	Idle-11 [EventBroker...
1:26.048	Dec 14, 20...	8030 µs	Blocked	104 bitronix.tm.resource.common.XAPool	Idle-9 [EventBroker th...	Idle-11 [EventBroker...
1:26.057	Dec 14, 20...	15 ms	Blocked	104 bitronix.tm.resource.common.XAPool	Device Result Process...	Idle-9 [EventBroker t...
Total:		372 ms				

Recording thresholds: **1,000 µs blocking / 100,000 µs waiting**

Filtered stack trace for waiting thread:

Filtered stack trace for owning thread:

Current Locking Graph | Current Monitors | Locking History Graph | Monitor History | Monitor Usage Statistics

Dec 14, 2011 10:06:21 AM 33:07

Looking again at the total time, at the bottom of the Duration column, you can see that a total of 372 milliseconds were spent blocked on XAPool during the workload. Compared to ~76 seconds in BTM-2.1.2.

This is not because of "lock duration" but because of less lock contention. Compared to BTM-2.1.2 where there were 26269 lock contentions in XAPool, in BTM-2.2 there were only 21.

Conclusion

To be clear, the difference between BTM-2.1.2 and BTM-2.2 is dramatic for my company's product. Bitronix went from a component that often showed up high on the radar when profiling our overall application (both in CPU and lock-contention), to a component that is largely transparent to us. Before, we had to "filter out" Bitronix when searching for performance problems in our application. With BTM-2.2 that is rarely necessary.

Having seen the gains in BTM-2.2, it would be hard to "go back" to the current release version. As it stands we intend to use BTM-2.2 in our next product release.