



Optimistic Thread Concurrency

WHITEPAPER

BREAKING THE SCALE BARRIER

Written by Brian Goetz

Azul Systems, Inc.
January 2006
AWP-011-010
Version 1.0

Copyright © 2006 Azul Systems, Inc. All rights reserved.



EXECUTIVE SUMMARY

As processor counts increase, standard approaches to concurrency control – mutual exclusion locks – can create scalability bottlenecks that lead to underutilized CPUs. To address the scalability problems of standard approaches to locking, Azul Systems[®] implements an innovative new locking approach called “optimistic thread concurrency.” Based on proven techniques used in database systems to improve throughput in transactional systems, optimistic concurrency improves throughput by allowing code that would otherwise be serialized to be safely executed concurrently.

When a multi-threaded application is executed on a small number of processors, concurrency bottlenecks often remain hidden. Applications that were not developed or tested on larger systems often exhibit poor scaling beyond a handful of processors, preventing them from taking advantage of additional compute resources. Through an advanced combination of hardware and virtual machine features, the Azul Systems' solution allows applications to overcome concurrency obstacles – without requiring customers to modify existing code.

The Java™ programming language provides constructs – such as synchronization – for creating thread-safe programs, but this safety comes at a cost. When multiple threads access a shared resource protected by a lock, all but one of them must wait until the thread holding the lock releases it. Using exclusive locks to protect frequently-used resources can have a significant impact on scalability. In larger multiprocessing environments, processors can remain idle even though there is plenty of work to do, because too many threads are blocked waiting to acquire the lock. More scalable software engineering alternatives to exclusive locks, such as read-write locks or non-blocking algorithms, are difficult to use, and it can be cost-prohibitive to modify existing applications.

If applications could overcome lock contention and other scaling barriers, customers could consolidate small application tier servers onto a larger system, reducing complexity and management overhead. Optimistic thread concurrency (OTC) does just that – improving scalability of applications without rewriting them. A single Azul compute appliance features up to 384 processor cores, and the Azul virtual machine environment can scale to support hundreds of active, concurrently executing threads. While multi-threaded applications would usually exhibit poor concurrency with so many processors, the Azul solution overcomes traditional lock contention issues to allow applications to exploit the computing power of much larger virtual machines.

LOCKING: ON A COLLISION COURSE WITH AMDAHL'S LAW

Amdahl's law tells us that throughput and utilization is limited by the portion of the total workload that must be serialized. Applications that run well on a small number of processors often suffer from serialization due to lock contention as more processors are added. In most JVM implementations, synchronized code protected by a given lock must be serialized, which can create a significant obstacle to effective scaling. Using a combination of hardware and software, Azul's Optimistic Thread Concurrency (OTC) technology can identify code paths that can be safely executed concurrently, even when synchronization would typically prohibit concurrent execution.

Figure 1 illustrates the effects of Amdahl's law. Each curve represents a program with a different degree of serialization. The graph clearly demonstrates how even a small degree of serialization can greatly impair a program's ability to exploit many processors. With 400 processors, a program with 1% serialization can use those processors with only 20% efficiency; a program with 5% serialization can at best only achieve 7% efficiency. Throwing more processors at the problem does not always yield the desired throughput improvement.

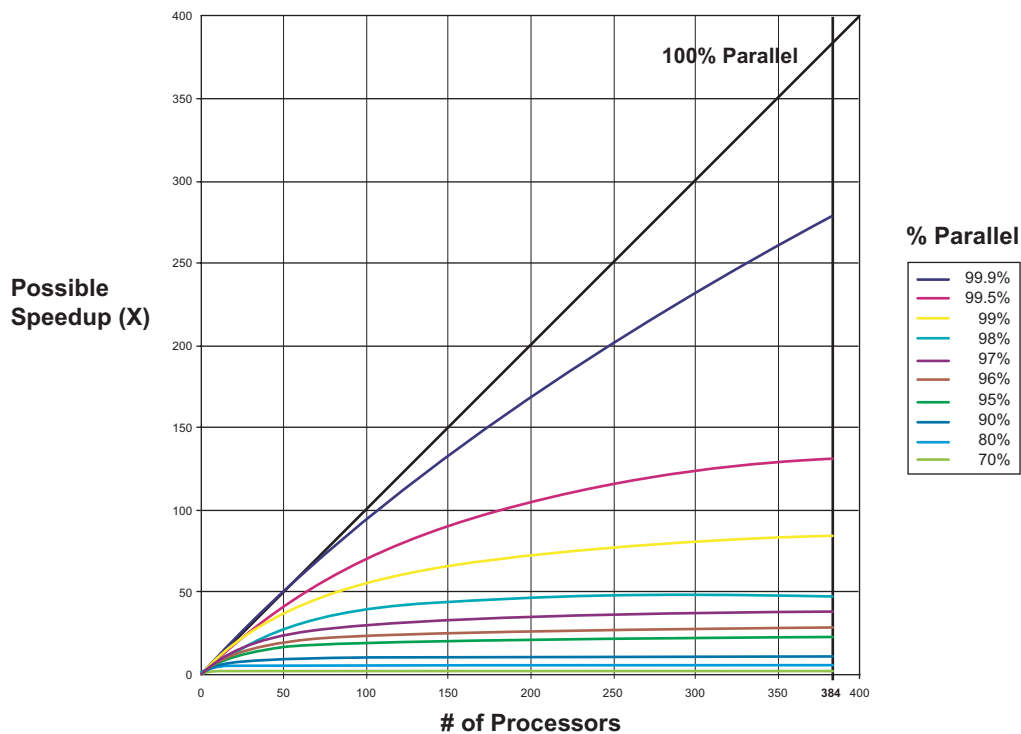


Figure 1 Amdahl's Law: CPU utilization as processor count increases for varying degrees of serialized code

When data in a Java program may be accessed by multiple threads, it is necessary to use synchronization to coordinate access to that data, so that threads do not modify data that another thread may be in the middle of using, and that they have access to the most up-to-date values for shared data. Using synchronization to ensure exclusive access to shared resources renders the otherwise asynchronous interactions between threads predictable. Nearly all Java applications and libraries rely on synchronization to ensure correct results in the face of concurrent execution.

The standard implementation approach taken by traditional JVMs is to treat synchronized blocks as requiring an exclusive lock, which has the effect of serializing access to the code blocks protected by each lock. This protects threads from unwanted interference from other threads, but on systems with many processors, the standard approach to concurrency control in Java programs appears to be on a collision course with Amdahl's Law.

LOCK CONTENTION VERSUS DATA CONTENTION

Exclusive resource locks can be a scalability hazard because if two threads want to access the same data, they will contend for the lock, and JVMs deal with lock contention by suspending all but one of the contending threads – serializing access to the resources protected by the lock. The longer that a thread holds a lock, the more likely it is that lock contention will occur.

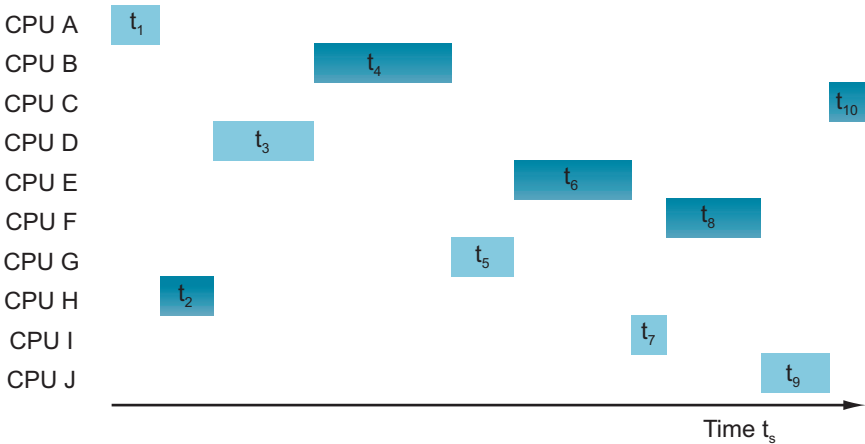
Programs don't acquire locks because they have any direct need for the lock; they acquire locks because they need access to the data protected by that lock, and locking is used to ensure that two threads don't manipulate the same data at the same time. But just because two threads contend for the same lock doesn't mean that they are contending for the same data. A lock may protect multiple variables, such as the entire state of an object or a group of objects. Threads that access protected data concurrently do not necessarily conflict with one other, such as two threads that want to access two independent variables protected by the same lock, or who both want to read the same variable but not write it. In many cases, exclusive locking is excessively conservative, leading to serialization of operations that would be safe to execute concurrently.

The reason that existing JVMs force operations to execute serially that could safely execute concurrently is that they have no mechanism for detecting data contention, making serialization the only way to preserve application integrity. Consider a thread-safe data structure containing product data in an e-commerce application, like the `Hashtable` class from the Java class library. It is likely that such a table would be infrequently modified, since product information does not change very often. In such a case, nearly all access to the product table by the program will be read-only, and many threads might want to access the data at the same time. Reads could outnumber writes by a factor of hundreds or thousands, making the "one thread at a time" locking discipline costly in terms of performance. If we knew that a thread requesting access to the lock only needed it for reading, we could allow multiple readers to proceed simultaneously, greatly improving throughput.

Even if writes were more frequent, a writer modifying one hash chain might not need to prevent readers or writers from accessing another. We could even allow readers to access the product table at the same time as writers, if we knew they were not accessing the same portion of the data structure, or even allow multiple writers to proceed concurrently if they were not touching any of the same data.

Using an exclusive lock to protect the product table ensures that the reader threads see the correct data values in the table and prevents the table data from being corrupted, just in case a rare writer makes any modifications. But exclusive locking requires all threads to access the data structure serially, even when some of them could safely execute concurrently. In this example, which is typical of many concurrent applications, threads are frequently competing for the lock, but only infrequently competing for the data. Most JVM implementations make no distinction between lock contention and data contention, and as a result, synchronization is implemented conservatively, causing serialized execution where it is not strictly necessary. Figure 2 illustrates the throughput cost of serialization, comparing serial versus concurrent execution of 10 application threads.

Serial Thread Execution



Concurrent Thread Execution

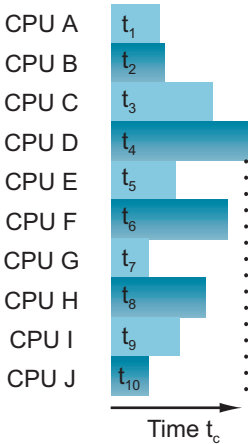


Figure 2 Throughput and utilization effects of thread concurrency

Lock contention could be reduced by rewriting the software. Software techniques like read-write locks, lock striping, and non-locking algorithms are well understood and can reduce the impact of lock contention, bringing locking patterns more in line with data access patterns. However, code often lives longer than we expect it to when it is written. Many Java applications running today were written when “lots of processors” meant no more than a dozen. These programs did not use these advanced techniques because they were not needed on the targeted deployment hardware. Rewriting existing code to employ advanced concurrency control can be costly, and these techniques may still not perfectly match locking behavior with data access behavior, so some operations may still be serialized where they could safely execute concurrently. Azul has a better approach – use a combination of hardware and software to detect data contention instead of lock contention, and execute operations concurrently when it is safe to do so.

OPTIMISTIC CONCURRENCY – BORROWING A PAGE FROM DATABASE HISTORY

Database systems promise that each transaction will be atomic and isolated, requiring the database to preserve the illusion that every transaction is the only transaction running. Of course, database systems can execute transactions concurrently if it is safe to do so. Early database systems used locking at the table, page, or row level, but today many database systems also use a more scalable technique called optimistic locking. Optimistic locking can be compared to the old saying that “it is easier to obtain forgiveness than to obtain permission.” With optimistic locking, the database allows multiple transactions to proceed concurrently, but rolls a transaction back if it detects that any of the data that it has worked with has been modified by a concurrently running transaction. This amounts to detecting data contention at commit time, rather than preemptively serializing transactions that threaten to operate on the same regions of data, resulting in greater throughput.

The Azul implementation of optimistic thread concurrency is similar in design to optimistic locking, preserving atomicity for synchronized blocks in Java programs while providing the scalability benefits of increased concurrency. Like optimistic locking, OTC makes the assumption that, although lock contention may occur frequently, data contention occurs far less frequently. OTC permits many operations that would otherwise be serialized to execute concurrently, which can greatly enhance application scalability.

To maintain integrity, OTC in the Azul VM employs a combination of hardware assists and software techniques to detect data contention as it occurs. The Azul VM can grant a lock to a thread speculatively; if it detects that another thread has written data that is was speculatively read, or read data that was speculatively written, data contention is deemed to have occurred. Contention is also assumed if operations that cannot be tracked for contention, such as performing I/O, are done while speculatively holding a lock. When contention is detected, the Azul VM transparently “rolls back” any effects of code executed since the lock was speculatively taken, and the thread transparently resumes execution at the lock acquisition point. The application code is never aware that data contention or rollback have occurred.

OPTIMISTIC EXECUTION OF SYNCHRONIZED BLOCKS

The essence of multi-threaded synchronization in Java is that synchronized blocks be executed atomically in relation to other blocks synchronizing on the same object. JVM implementers have traditionally interpreted “synchronized” to mean “acquire the lock, execute this block of code, and then release the lock.” While this interpretation certainly provides the required atomicity, it is much more conservative in nature than the specification requires. In fact, the requirement can be conservatively met by promising to “execute this block of code atomically with respect to all other synchronized blocks.” This is equivalent to promising that synchronized blocks be executed transactionally.

To execute synchronized blocks transactionally without serializing all execution of synchronized blocks, the JVM would have to detect contention for data with other transactions, and roll back any transactions for which data contention has been detected. So long as these guarantees are preserved, synchronized blocks, even those protected by the same lock, can be allowed to proceed concurrently. And this is what OTC does – through a combination of VM techniques and hardware assists, it allows multiple synchronized blocks to execute concurrently, but speculatively; if data contention is detected by the hardware, the VM rolls back the effects of the speculative execution.

The scalability improvement can be dramatic. Uncontended synchronized blocks execute just as fast as they would with serialized locking. Synchronized blocks that actually contend for the same data are executed serially, but synchronized blocks without data contention can execute concurrently, even when protected by the same lock. Amdahl’s law still governs the scalability of the application, but the meaning of “serialized” has been redefined. Instead of serializing operations protected by the same lock, only operations that actually contend for data are serialized. In typical Java applications, this means greatly reduced serialization, and therefore enhanced scalability.

ADAPTIVE SELECTION OF LOCKING STRATEGIES

OTC uses a speculative locking mode that can detect contention for data and roll back the effects of the synchronized block. But the JVM can do even better, by not using speculative locking when past profiling information suggests that data contention is likely. Similarly, it can use “thin” locking, a strategy employed by many JVMs when profiling information suggests that lock contention is unlikely, and use the pessimistic “thick” locking strategy when it thinks both lock and data contention are likely.

- **Thin locks** are used when there has been no contention for the lock. Thin locks are the most efficient form of locking. A thin lock that experiences contention must then change to a speculative or thick lock.
- **Thick locks** are full-blown mutual-exclusion locks that imply serial execution. When contention occurs on a thick lock, the contending threads serialize on the lock and block until the lock is released.
- **Speculative locks** are the essence of OTC. When the Azul VM selects speculative locking, multiple threads are allowed to continue past the lock acquisition and execute simultaneously while the hardware monitors for data contention. If the hardware detects data contention, then the JVM rolls back the affected threads to the beginning of the lock, erasing all effects of speculative execution.

Whenever a thread attempts to acquire a lock, the Azul VM makes a decision as to which locking strategy it should use: traditional (thick) locking, thin locking, or speculative locking. As it gathers data about lock and data contention patterns, it can make better decisions, choosing the optimal strategy on a lock-by-lock basis. In many cases, it can acquire the necessary profiling information in the first few seconds of run-time, allowing the JVM to make highly intelligent decisions about the best locking approach very quickly. If speculatively acquired locks need to be rolled back too often, the VM will recognize the speculative locking is not appropriate for this lock, and fall back to thick locking and serial execution so as to avoid the overhead of frequent rollback. In a great many cases, threads contending for the same lock do not contend for data, and speculative locking results only in greater scalability. The speculative locking mode would work very well in the previously described example of a Hashtable containing product data. Since reads greatly outnumber writes, data contention is extremely rare, and speculation under optimistic concurrency will almost always be successful. Even if writes were more frequent, threads that access different hash chains might (depending on the Hashtable implementation) be able to safely access the data structure concurrently.

Figure 3 demonstrates the performance improvements of OTC, showing the throughput of an application where multiple threads repeatedly access a Hashtable under traditional (thick) locking and speculative locking; with OTC, throughput is more than doubled with a large number of threads.

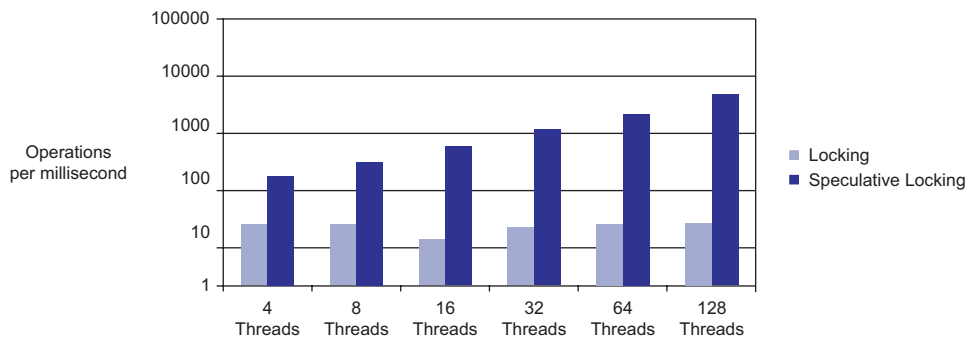


Figure 3 Measurements: hashtable (5% writes)

In this example, the threads read from the Hashtable 95% of the time, and write to it 5% of the time. Some percentage of the time the speculation will fail and need to be rolled back, but this is more than made up for by the increased concurrency when optimistic locking allows multiple operations that would otherwise require serialized execution to successfully complete concurrently. If the percentage of threads requiring write access was lower, as it would be in the product data example described above, the throughput advantage of OTC would be even greater.

CONDUCTING ROLLBACK

In the event that speculative execution fails, the Azul VM must safely and transparently perform a transactional rollback. To support this rollback capability, the VM maintains the state of the speculating thread at the beginning of each speculation, and assures that none of the thread's actions are visible to other threads until speculation is complete. If contention is detected, the thread state is restored to the saved pre-speculation state, memory changes are reverted, and thread execution resumes at the beginning of the synchronized block.

Rollback is transparent to the application, and no code changes are required to support speculation or rollback. To reduce the need for rollback, the VM uses adaptive selection of locking modes. When speculation fails, the VM updates profiling information associated with each lock, and will not select speculative execution if speculative locking for this lock has failed frequently in the recent past.

CONCLUSION

Network attached processing enables a new paradigm to support Java-based enterprise applications. This new approach executes the entire virtual machine workload to dedicated processing power in the Azul compute pool. The Azul solution permits commercial Java-based applications to overcome many previous hardware and software constraints – allowing applications to scale and take advantage of the tremendous processing power of the Azul hardware while delivering the benefits of lower cost and reduced management complexity for the application tier.

One factor that can inhibit concurrency and the scalability of Java-based applications is the impact of Java synchronization. Many Java libraries and Java-based applications implement locking to ensure exclusive access to data. Often, these locking mechanisms result in serialized execution – even if no data contention ever occurs. Azul's adaptive locking strategy supports a speculative locking technique called optimistic thread concurrency, which overcomes the excessively conservative nature of locking and manages concurrency on the basis of data contention, not lock contention. Optimistic thread concurrency preserves atomic execution of synchronized blocks while allowing greater concurrency, improving scalability while guaranteeing application and data integrity.

ABOUT THE AUTHOR

Brian Goetz has been a professional software developer for the past 18 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, CA, and he serves on several JCP Expert Groups. See Brian's published and upcoming articles in popular industry publications, and look for his upcoming book, *Java Concurrency in Practice*, in June 2006 from Addison-Wesley.

ABOUT AZUL SYSTEMS

Azul Systems has pioneered the industry's first network attached processing solution designed to enable unbound compute resources for Java and J2EE based enterprise applications. Azul compute appliances eliminate capacity planning at the application level and much of the cost and complexity associated with the conventional delivery of computing resources. More information about Azul Systems can be found at www.azulsystems.com.

Copyright © 2006, Azul Systems, Inc. All rights reserved. Azul Systems and Azul are registered logos in the United States and other countries. The Azul arch logo, Compute Pool Manager, and Vega are trademarks of Azul Systems Inc. in the United States and other countries. Sun, Sun Microsystems, Solaris, J2EE, J2SE, Java and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Other marks are the property of their respective owners and are used here only for identification purposes. Products and specifications discussed in this document may reflect future versions and are subject to change by Azul Systems without notice.

