追求原始速度的过程，会使可读性强的代码变得晦涩难懂（通常是无用的——目前的优化编译器相当不错），从而导致以后维护困难；而且在很多情况下，整体方案中的优化性能指标无法吸引人们的兴趣。很多人只注重完成一次请求所需的时间——“这次交易仅用了 20 微秒就从储蓄帐户中取出 500 英镑，哇！”

对于事务型系统来说，系统的吞吐量通常远远要比一次请求的绝对速度更吸引人（尽管这么说，执行事务所需的时间还是必须满足某种限制条件）。吞吐量衡量系统在达到响应时间目标的情况下可以处理的工作量。当然了，系统中更多的客户端请求事务是产生更多工作的部分原因。此处的另一个有趣的因素是，响应时间和客户端数量不是独立的变量——抛给系统工作的客户端越多，单个事务就越有可能使用更长的时间。因此，（在给定的部署环境下）系统的最大吞吐量支配着多少个客户端可以以某个预期的速率提交事务，并将多少事务（比如说 90%）的响应时间维持在要求的时限内。得出这个结论以后，在尝试预测生产设置以满足所需的服务水平并优化服务器资源的使用时，更改各种系统参数（执行线程数、数据库连接数、机器数等等）会不断地产生有趣的结果。

事实上，对于任何使用应用服务器的人来说，单个事务的个体往返时间对性能没有太大影响已经不是什么新鲜之谈了。很明显，优化的方式是尽可能地缩短客户端和该客户端所作用的后端之间的代码路径——而客户端和后端之间的应用服务器基础架构中的粘合层显然不利于缩短代码路径。但是，它却能提高吞吐量，这主要是通过在客户端中间共享稀有的服务器端资源（数据库连接、线程等等）而实现的。如果简单地通过缩短代码路径来提高性能，最终只会在客户端数目和服务器上所使用的资源之间建立一对一的关系；而一旦所有的服务器资源都被使用了，就会产生性能瓶颈，吞吐量也就无法进一步得到提升，除非可以整合更多的资源并共享出来。应用服务器的真正作用是，通过在客户端之间共享来节约服务器端所使用的资源数（代价是会产生较长的代码路径，这暗示着会或多或少地延长事务的往返时间），从而提高系统的最大可能吞吐量。从这方面来说，“我不使用事务，它们会降低速度”的说法是有道理的。

到目前为止，我们已经看到，系统要在有限的服务器资源下运行尽可能多的请求。但是没有说所有这些请求必须得到正确的结果——也就是说，任何独自执行的请求的结果应当与它与其他许多请求一起执行时所得到的结果一致（或者说，请求应当具有隔离性，当然了，隔离性(isolation)就是 ACID 中的“I”，而提供 ACID 属性的是事务）。因此，使用 XA 事务的代价是牺牲一点点绝对性能，而益处是得到正确的事务结果。如果您存款的银行使用 XA，那么您就可以高枕无忧了，因为您知道，即使在某些模糊负载条件下从银行帐户中提款也不会出错！

这些讨论全都发生在数据层——XA 完全是关于数据库事务的。每一个事务会将数据锁定在数据库中，直到事务完成；随后等待方才能看得到结果。当多个请求试图访问相同的数据时，事务就

会引发瓶颈问题——除了第一个锁定争用数据的请求之外，其他请求都被阻塞或抛出，从而严重地影响了吞吐量，因为正在做的工作不会导致系统中有一个良好的事务流。

然而，数据库中的数据不是事务系统中唯一共享的资源。我已经讨论了应用服务器作为资源共享机制的作用——任一个资源都可能被争用；因此，如果多个请求同时访问，则应用服务器本身需要锁定内存中的数据结构以避免产生问题，而且这里也会产生争用。当然了，除应用服务器之外的各层也进行资源共享——在一个典型服务器中，为 WebLogic 配置的 60 个执行线程很可能至多在几个 CPU 中执行——当发生 CPU 争用时，那些不走运的线程就必须在队列中等待，直到某个 CPU 空闲；内存或者硬盘也是如此。

总之，到目前为止我所提到的都是大问题——性能测试的开发、完成所有调优后测试的运行、满足预期要求的服务器资源分配，这些都随着不断变化的应用程序版本和服务器环境版本而反复进行，产生的特定系统必须承受得起不可预知的实际负载。这些不只是难题，也是每一个应用程序生命周期成本的主要部分。

聪明的读者会注意到我的电子邮件地址已经变更；这是因为我已经加入了 Azul Systems，该公司有应对上述问题的独特解决方案。Java 是多线程的，因此可以提供包含多个 CPU 且每个 CPU 都具有多个内核的系统；这些 Java 线程实际上就产生了并行化；其次，它提供对同步 Java 对象的乐观锁定的支持，以减少 Java 软件层中的争用；再次，它提供可以在多个应用程序之间共享的内存池。总而言之，Azul Appliance 提供了一种 Java 执行引擎，它通过为单个应用程序提供大量的 CPU 和内存资源，从而试图缓解传统虚拟机环境中由于 CPU 或内存不足而产生的瓶颈，从而降低了系统使用的资源量。

该策略还有另外一个优点，即它在一定程度上避免了基于每个应用程序进行调优的需要，这是因为设备提供的资源非常多，以至于可以轻松地在多个应用程序之间共享；而以前，每一个应用程序都需要在自己的服务器上为需求峰值预留一定的空间。这种共享是可以实现的，因为总的来说，需求峰值是平滑的（因为很多不相关的应用程序同时达到需求峰值的情况不大可能发生）；因此，需求可以通过池化的资源得到满足，从而使您既不用担心应用程序级的大小（以及相关成本），也不用担心每个应用程序基础上的硬件过量供应。这预示着一个以低成本采购和管理的设备来运行高性能 Java 应用程序的新时代来临了。

# WebLogic Performance: Pursuit of Speed Isn't Everything

*"High performance" is what everybody strives for when putting together a new system. Technical folk often spend hours hung up on the raw speed of their code, and a certain machismo can be derived from shaving milliseconds off that pesky transaction that is the latest pride and joy. Often, this time is not very well spent.*

In the pursuit of raw speed, not only does readable code often get substituted for the obscure (usually to no avail - optimizing compilers are pretty good these days), thus causing a maintenance headache in the future, but in many cases the performance metric being optimized is a pretty uninteresting one in the overall scheme of things. Many people develop a blinkered focus on the amount of time it takes for one request to do a round-trip - "That debit transaction just took 20ms to debit £500 from a savings account, whoopee!!"

For a transactional system, the throughput of the system is generally far more interesting than the absolute speed of a request (although, that said, there is mostly some constraint that must be met around the time it takes to execute transactions). Throughput is a measure of the amount of work that can be driven through the system while achieving the targets for response time. More work, of course, is partly driven by more clients requesting transactions in the system. The bonus amusement factor here is that the response time and the number of clients are not independent variables - the more clients throwing work at the system, the more likely it becomes that the individual transactions will take longer. Thus, the maximum throughput of a system governs (for a given deployment environment) how many clients can throw transactions into the system at some desired rate, keeping the response time of some fraction of the transactions (say, 90 percent) within the required limit. Having arrived at this conclusion, hours of fun ensue changing various system parameters (number of execute threads, number of database connections, number of machines, etc.) in attempting to predict what the production setup needs to look like to meet the required service levels and to optimize use of server resources.

In fact, it should not be at all surprising to anyone who is using an application server that the individual round-trip time of a single transaction is of little interest. Clearly, the way to optimize that is to have as short a code path as possible between the client and the back end it affects - sticking layers of application server infrastructure

between the client and the back end is clearly not going to shorten the codepath. It does, however, improve the throughput. It does this mainly by sharing scarce server-side resources (database connections, threads, etc.) among the population of clients. With the simple-minded short codepath route to performance, you will end up with a one-to-one relationship between the number of clients and the resources consumed on the server, and once all the server's resources are consumed, you have reached a performance bottleneck and the throughput can be increased no further until you can scrape together some more resources to share out. What the application server is doing in effect is economizing the number of resources consumed on the server side by sharing them out among a population of clients (at the expense of a longer codepath, implying marginally increased transaction round-trip time), thereby increasing the maximum possible throughput of the system. At this point, everyone's favorite knee-jerk reaction: "I don't use transactions, they slow things down" can be usefully reviewed.

By now we have seen a system rushing as many requests as possible through a limited set of server resources. It goes without saying that all of these requests must achieve the correct results - that is, the result of any one of the requests executing on its own should be the same if the request is executed in parallel with lots of other requests (or the requests should be isolated, to use the parlance and yes, isolation is the "I" in ACID, and transactions are what give you ACID properties). So the cost of using XA transactions is a little absolute performance, but the benefit is that the results of your transactions are correct. You can rest easy at night knowing that withdrawals from your bank account aren't going to go screwy under some obscure loading conditions IF your bank is using XA!

This discussion takes place completely at the data level - XA is all about database transactions. Each transaction locks the data in the database until the transaction is complete, whereupon the results become visible to the waiting world. Transactions cause bottlenecks when multiple requests are trying to access the same data - all but the first locker of the contended data get blocked or thrown out, which clearly hurts throughput since work is being done that will not result in a successful transaction flowing through the system.

However data in the database is not the only shared resource in a transaction system. I already talked about application servers being resource sharing mechanisms - any of these resources could be contended for, so the application server itself needs to lock in-memory data structures to avoid problems if they are accessed on behalf of multiple requests in parallel, and contention can result here too. Of course, layers other than the app. Server are resource sharing too - on a typical

server, those 60 execute threads you configured for WebLogic are probably being executed across a handful of CPUs at the most - when contention for CPUs occurs, the unlucky threads must wait in line until a CPU is free and so on with memory, disk, etc.

Ovrall, everything I have thus far mentioned represents a massive headache - the development of the performance tests, the running of the tests while all the tuning is done, the allocation of server resources to meet the predicted demands, and all of this is repeated with changing application releases and server environment releases and the resulting sized systems then have to stand up to unpredictable real-life loading. Not only is this a headache, it is a major lifetime cost of every application.

The astute among you will have noticed that my e-mail address has changed; that's because I have joined Azul Systems, who have a unique solution to these issues. Java is highly multithreaded, so it provides systems that contain many CPUs, each with many cores so those Java threads really result in parallelization. Then it provides support for optimistic locking on synchronized Java objects, to reduce contention within the Java software layer, then it provides pools of memory that can be shared between multiple applications. In short, the Azul Appliance provides a Java execution engine that dwarfs your systems' capacity to consume resources by overwhelming your individual applications with CPU and memory resources in order to try and alleviate any bottlenecks caused by the shortage of either in the traditional VM environment.

This tactic has the additional benefit that it more or less removes the need to do tuning on a per-application basis, since the set of resources provided by the appliance is so big that it can be readily shared among multiple applications that previously each required headroom on their own servers for peaks in demand. This sharing is possible because on aggregate the demand peaks will be smoothed (since it is improbable that many unrelated applications will all hit a demand peak at once), so demand can be met from the pooled resources, which leaves you free not only from application-level sizing (and the associated costs), but also from large-scale overprovisioning of hardware on a per-application basis too. All this heralds a new era of cheaply procured and managed appliances to run highly performant Java applications. For more details, visit www.azulsystems.com. That's all for this month; watch out for a repeat performance next month.