



Azul's Assiduous Advance

Research Note

Gordon Haff

29 June 2007

When Azul Systems emerged from stealth mode in 2004, we were intrigued but skeptical.¹ The company's Java "Compute Appliance" was latching onto all the right technology trends. It was ahead of the curve in its wholehearted embrace of massively multi-core processors capable of handling many streams of instructions at the same time. It focused on "managed code," especially Java, running in a virtual machine independent of the underlying processor architecture—a direction in tune with the decades-long progression of software to higher and higher levels of abstraction and Java's increased usage in even very high-end transactional roles. And it arrived as an appliance, better to slip into existing datacenters and applications. So in many respects Azul was a new systems company with a new processor architecture, but one that didn't carry the same implications of *Rewrite All Software!* that it would have historically.



At the same time, we weren't wholly sold. Azul was flying in the face of the last couple of decades of computer industry history—bucking the trend toward systems that are more general purpose and "commoditized,"² to say nothing of a system vendor market in which it seemed harder and harder for small players to survive, much less prosper. We had technical concerns as well. When offload speeds up only part of a task, its effect on overall throughput can end up being quite small, even if it speeds up that one part considerably.³ Thus, even if Azul could, in fact, do wonderful things for Java processing, would it really boost the performance of the application environment as a whole enough to make it interesting to potential buyers?

Licensed to Azul Systems, Inc. for web posting. Do not reproduce. All opinions and conclusions herein represent the independent perspective of Illuminata and its analysts.

Now that Azul's into its second generation of product and has a broadening set of customer experiences and results to discuss, we're significantly less skeptical about

- ¹ See our *Azul: A New Shade of Server*. This note provides additional background on the company and the technical concepts behind its Compute Appliance.
- ² An overused term to be sure. Nonetheless, there can be no disputing that today's plethora of dual-socket servers is far more similar than systems of years past.
- ³ An example of Amdahl's Law, named after computer architect Gene Amdahl, which quantifies the limits to processing speedups when only part of the task is affected.

the technology and the company. Market trends such as server consolidation—which was much of the initial impetus behind server virtualization—and an increased attention on the part of customers on power-efficient processing are a good match with what Azul is selling. The company itself is better understanding the best places to apply its technology. And, perhaps most important, we're seeing specific examples of people who are using the gear in real environments—and liking it. In many respects, Azul is still in relatively early days, but it's no longer merely a startup with an intriguing hyperbolic pitch.

A New Type of Appliance

Azul CEO Stephen DeWitt is no stranger to appliances. He founded Cobalt Networks, which he later sold to Sun for \$2 billion during the dot-com boom. Cobalt was probably the most visible of the many companies creating single-purpose devices during this first Internet wave. The idea was that you could just plug an appliance into the network and use it for Web serving or just about any of a variety of common point functions. In practice, however, it proved hard to find the right blend of simplicity and flexibility, and to make the overall computing environment easier to manage. For these and other reasons, server appliances never really lived up to their promise; while successful in storage and security, they largely faded away as a server category.⁴

A major factor militating against those first-generation appliances is that they were largely software and packaging plays, rather than true systems plays. Most appliances were just general purpose x86 servers of one sort or another—usually running some variant of Linux and other Open Source software with a custom Web-based front-end. Thus, while they ostensibly offered simplification compared to general-purpose servers, they weren't otherwise better. And, for the most part, users found that adding appliances from a passel of vendors could actually make their hardware environment more diverse and

complicated. It's often better to deal with software complexity by other means—such as provisioning software or, more recently, virtual appliances.⁵

Thus, it's significant that Azul's Compute Appliance isn't just a general-purpose server beneath the covers, and that it doesn't just tweak general-purpose software for its specialty. It is based on customized hardware and bespoke software—however contrary to current industry trends that may be. An appliance that represents a complete system optimized for a specific task is much more differentiated than one that's largely a packaging of mostly off-the-shelf piece parts.

The Concept

Azul's approach is conceptually simple. It provides a network-attached "appliance" running Java Virtual Machines (JVM)⁶—typically one per application—onto which other servers can transparently offload their Java processing. Essentially, it's a compute resource that other computers on the network can share. It operates on data in memory, but doesn't store it in any permanent way. Azul has specifically designed its appliances to offload Java processing, as opposed to arbitrary program binaries—depending on the way in which Java genericizes its execution engines and environment as a fundamental characteristic. It also depends on the fact that Java, rather than traditional C and C++ code, is at the center of today's new software development.⁷

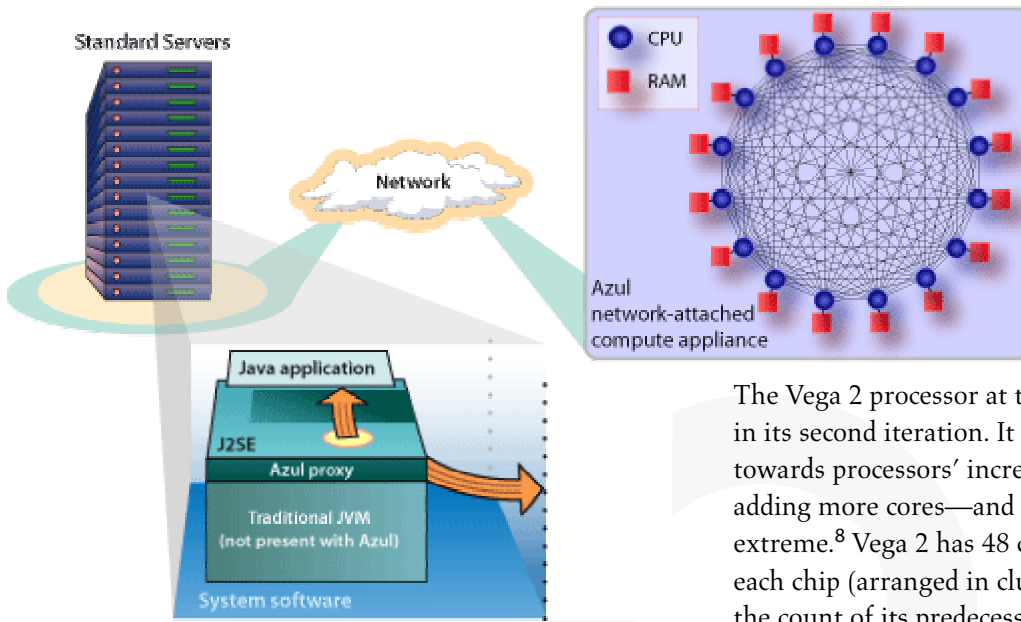
In order to access Azul's Compute Appliance, "application host" machines (i.e. general purpose computer systems) load the Azul Virtual Machine,

⁵ See our [Bemoaning Laggardly Virtual Appliances](#) and [Virtual Appliances Evolve](#)

⁶ JVMs are sometimes categorized as "application virtual machines." This distinguishes them from system virtual machines (VMs) in the vein of IBM's z/VM, Xen, or VMware's ESX Server, which run a complete operating system image in each VM.

⁷ In principle, Azul could handle .NET managed code in a similar way, but is concentrating on Java for now—that being the more common enterprise software environment in the financial and other industries on which the company is focusing its sales energies.

⁴ See our [The End of Cobalt and the Appliance Era that Never Was](#)



University of California, San Diego.

The Implementation

The Azul system hardware itself is a largely custom design that is optimized for Java-style workloads.

a fully-compliant JVM implementation. The Java application is then launched on the Azul Virtual machine (instead of a conventional JVM) that then sends the Java bytecode and application data over the network to execute on the Azul compute pool. Application servers like BEA WebLogic or IBM WebSphere point to the VM simply by setting the right path in a configuration file or at the admin console. However, instead of starting a local JVM, as would normally happen, the Azul software starts a virtual machine proxy. This proxy redirects the Java workload to an Azul compute pool (i.e. one or more compute appliances). To outside systems it appears as if the JVM is still executing on the application host. Even the application itself cannot tell that it's executing externally; it just invokes the `java` command in the usual way.

This all works transparently because a JVM is software that creates an abstract computing machine, an environment that runs on top of a real physical system. Code running within a JVM sees the same instruction set and execution environment no matter what actual underlying processor and OS are being used. Thus, the same code that runs on a JVM for Linux/x86 will run on a JVM for Solaris/SPARC. This “write once, run everywhere” concept has a long history; the “p-Code machine” used by UCSD Pascal is a well-known historical example developed in 1978 at the

The Vega 2 processor at the heart of things is now in its second iteration. It follows the general trend towards processors’ increasing performance by adding more cores—and takes that trend to an extreme.⁸ Vega 2 has 48 coherent 64-bit cores on each chip (arranged in clusters of eight)—doubling the count of its predecessor. By contrast, Sun’s UltraSPARC T1 (“Niagara”), which is at the forefront of ratcheting up core counts in general purpose computing, only has eight cores per chip.⁹ Two to four cores are the norm in most designs.

Vega 2’s L1 cache has 16 KB instruction and 16 KB data for each core. There is 1 MB of L2 cache shared by each cluster of eight cores, for a total of 6 MB on the chip. Cache coherency is managed by the hardware. Each processor also integrates four DDR2 memory controllers, supporting a total of 12 DIMMs. Using 4 GB DIMMs, memory scales at 1 GB per core—thereby allowing for appliances with truly gargantuan memory capacities to keep pace with the equally impressive core counts—768 GB for 768 cores in the largest configuration. The processor has 812 million transistors and is fabricated by TSMC using a 90nm process.

Each processor also has 15 “Xlinks,” enough high-performance interconnects to let each processor chip connect directly to every other processor in the system.¹⁰ This fully-connected mesh

⁸ See our [Breaking Up The Microprocessor Monolith](#).

⁹ Sun augments the cores with a multi-threading mechanism that lets each core quickly switch between threads. However, multi-threading doesn’t add as much performance as having additional full cores. See our [Gradations of Threading](#).

¹⁰ The technical name for this topology is “complete graph” or Kn. Crossbars with this many connection

interconnect coherently links up to 768 individual processing cores (on 16 chips) without the additional latency and lower performance that multiple stages of crossbar (which would require fewer links per processor) could introduce.

This second generation of Vega also builds on its predecessor's hardware assists for "garbage collection." This feature—in conjunction with the JVM software that works in concert with the hardware—is a major Azul selling point.

Taking Out the Garbage

In a language like C++, the programmer has to explicitly allocate memory for each object (whether a simple variable or something more complex) and then destroy it when it is no longer needed. In addition to taking a fair amount of effort, manual memory management is error-prone. For example, repeatedly creating objects without deleting them creates "memory leaks" that cause a program to use more and more memory as it runs—even to the point that there's no more free memory and the system has to be restarted. Other potential problems include "dangling pointers"—a pointer to a deallocated location of memory—which cause unpredictable program behavior and lots of debugging headaches.

By contrast, one of the attractions of languages such as Java, C#, Ruby, Perl, Python, and so forth is that they take care of memory management automatically.¹¹ In the case of Java, the JVM does this by storing all objects created by a running Java program on the "heap," a dynamically-sized chunk of memory. Rather than the programmer's then having to deallocate heap objects when they are no longer needed, the JVM does so automatically. It does this by keeping track of whether the program is still using a given object or not (in tech-speak,

points rarely connect to every other node as the Azul system does. In systems with fewer than the maximum number of processors, the "extra" Xlinks go unused rather than being configured for redundancy or extra bandwidth.

¹¹ Although automatic memory management is most associated with relatively modern languages, its origins go back to John McCarthy and LISP in 1960.

whether it's *referenced* or not), and freeing up the space if not.

This process of freeing up space is called "garbage collection" (GC). It's a way to automatically reuse memory that's no longer needed by a program. The idea is pretty straightforward—as are small-scale, low performance implementations. But handling garbage collection at scale is a much trickier proposition—and can ultimately limit the size of the heap and thereby the amount of memory that can be used by a Java program.¹² To be sure, the cases where this is a truly show-stopping problem will tend to be very large Java application environments with stringent service level agreements (SLAs). However, it's no theoretical problem; examples exist in places like Wall Street. Azul cites the example of a financial services application that had measured partial GC cycles of between 2 seconds and 70 seconds and a full GC cycle that took 4 minutes (during which other processing stops). The same firm had a required SLA of less than one second, a measure of the fact that for many financial transactions trades *must* go through quickly or the trading order will most likely be routed elsewhere.

In part because of its ability to do what it calls "Pauseless Garbage Collection," Azul says that it has seen 5x to as much as 50x performance and scalability improvements based on collected data from a variety of financial services firms such as investment banks, telcos (e.g. British Telecom), and other enterprises.¹³ In another example, Steve Lapekas, the CTO of Pegasus Solutions (which provides reservation and other services to hotels) told me that the massive memory available and the garbage collection capability on an Azul Compute Appliance have allowed him to aggregate previously segmented apps into a single JVM where they can address 10 to 15 GB of stack and

¹² The JVM specification requires that the heap be garbage-collected, but doesn't get into the hows.

¹³ Many of these are unnamed, but that's not particularly surprising given how financial services firms tend to play their technology cards pretty close to the vest for competitive reasons.

heap space—a more optimized configuration when performing searches, for example.¹⁴

Evolving the Company

Part of Azul's progress can be measured on the basis of product. The company has just released its 7200 Series systems, the high-end of its second generation lineup. We're also seeing more and more concrete proof points about that product. Many customers are still kicking tires or getting ready to do so. But, in addition, there are named and unnamed enterprises working with Azul's gear, and there is solid data from those customers. Evidence is mounting that the systems work as advertised when there's an environment of high-scale Java in which they can thrive.

Azul as a company is also evolving as it finds its feet as a system supplier and not just a startup. It's identifying the kinds of companies with which it can successfully work, rather than talking loudly about transforming computing as we know it. CEO Stephen DeWitt remains as enthusiastic as ever, of course.¹⁵ But there are now other, more nuanced, voices more focused on appealing to the enterprise buyer than the Silicon Valley VC. For example, the recently appointed VP of Marketing, Ram Appalaraju, was a long-time marketing exec at HP.

Such folks are starting to ask questions such as "What business apps can we go after and make a difference?" There's Java, obviously. But it's more specific than that. After all, any system can run Java workloads. Where Azul comes into its own is when an organization's application needs very large integrated object heaps—where large means into the tens or even hundreds of gigabytes and where SLAs are important (and therefore GC pauses are bad or even unacceptable). In other words, Azul's focusing on those environments where there aren't many—if any—good alternatives to its approach, and where, not incidentally, there's a willingness to spend for something that delivers real advantage. While early customers and testers span a range of

industries, finance and telco dominate. Though it obviously won't turn down business from other industries, Azul will be concentrating in specific verticals such as these that are a good match with its technology, and develop expertise and exploit beachheads there.

In addition to raw single-application vertical scale and the large data caches often associated with them, Azul sees itself as playing in a couple of other areas as well—both of which reflect an IBM mainframe pitch in many respects.¹⁶ One is as a platform for shared services; by running many services on a single virtualized system, it's easy to rapidly reallocate resources as workload needs change. The other is as an engine for Service Oriented Architectures (SOA); putting multiple services on one box can decrease the latency of the various services communicating and thereby make transactions more predictable.

Azul would seem to have fewer unique advantages in going after shared services and SOA opportunities and more operational inertia by customers to overcome. Other large SMP systems can leverage virtualization to make alternative runs at the problem. Furthermore, whatever the issues associated with managing large populations of small servers, distributed services are still more commonly associated with distributed systems' running all manner of workload management and performance optimization software. Sometimes referred to broadly as "grid," the vision of suppliers like Dell is to leverage the economics of volume manufacturing to run jobs across a large, coordinated group of off-the-shelf x86 servers.

Azul and other large system suppliers have to fight against the attractive hardware acquisition costs of volume servers. But the simplicity of a large, unified pool of compute resource, managed by the machine(s) and not the programmers, is on Azul's side. Furthermore, Azul has a feature called DirectPath that lets multiple JVMs consolidated on a compute appliance communicate directly without going through the TCP/IP stack and network—

¹⁴ See our [Azul Books a Room](#)

¹⁵ See our [Azul 2.0](#) for further discussion of Azul's evolution as a company.

¹⁶ See our [z/VM: Teddy Bears and Penguins](#) and [The Mainframe and the Four Happy Myths of SOA](#)

thereby decreasing latency relative to a typical distributed implementation.¹⁷ Another part of the answer is to work with such grids; for example, Azul counts software grid vendor Tangosol (recently purchased by Oracle) as a partner.¹⁸

Nonetheless, whatever the marketing and sales challenges associated with going after SOA and shared services workloads, it makes sense for Azul to do so precisely because they are more mainstream and more broadly applicable than the very largest individual Java workloads; they make sense as a broadening of Azul's customer base.

Conclusion

One doesn't hear any claims these days that Azul's approach "just doesn't work." Even competitors use words like "interesting" when Azul comes up—as it seems to with increasing frequency. In fact, many seem willing to concede that it really can capably handle truly massive Java workloads that can be difficult to deal with in other ways.

Then they go on to note that such workloads are but the thinnest slice of computing and that general purpose is the way to go. At this point, with Azul's

¹⁷ This is conceptually similar to the benefits that IBM System z HiperSockets and System p Virtual LAN bring to SOA applications.

¹⁸ See our [Tangosol Coherence Data Grid](#).

having largely demonstrated that it can handle the big jobs, this seems the most on-point remaining criticism. To turn the earlier discussion on its head, how many Java apps have heap sizes in the tens or hundreds of gigabytes and must live by stringent SLAs to boot? They're hardly a-dime-a-dozen.

That said, large classes of applications whose performance requirements are growing at faster than Moore's Law rates—that is, faster than performance increases coming from the business-as-usual (from our industry's jaded perspectives) doubling of transistor density every 18 months or so. To be sure, some of these applications are in areas like high performance computing that don't tend to be Java workloads. However, many are driven by consumer-facing Internet activity; for example, one of the problems that Pegasus faced was that users are getting more and more quotes for hotel prices (for which Pegasus doesn't get paid) without a commensurate increase in bookings (for which it does).

Admit it. You're doing it too—looking here there and yonder, again and again, for that perfect price or that perfect item. Maybe you'll check out those user reviews one last time. And what's the score of the Red Sox game? *Click. Click. Click.* Do you really want to bet that those application loads aren't going to keep going up? And in a big way?



Through subscription research, advisory services, speaking engagements, strategic planning, product selection assistance, and custom research, Illuminata helps enterprises and service providers establish successful information technology.