



The Azul Virtual Machine

WHITEPAPER

REALIZING THE ELEGANCE OF JAVA™ TECHNOLOGY

Written by Brian Goetz

Azul Systems, Inc.
January 2006
AWP-008-012
Version 1.2

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



EXECUTIVE SUMMARY

This paper describes the innovations in the Azul[®] Compute Appliance with a focus on the Azul Virtual Machine, including the new Vega[™] processor, the relationship between the Azul VM proxy and the Azul VM engine executing on the compute appliance, the synergies between the Azul hardware and the VM, and powerful new optimizations enabled by the unique features of the Vega processor. The paper is intended for engineers and system managers who are responsible for application architecture, development, and deployment.

The J2EE™ platform has been wildly successful in the enterprise – Gartner estimates that by 2008, 80% of all new e-business application development will be based on virtual machine technology. Many enterprises are running hundreds of J2EE platform based applications, with new applications being deployed daily. The availability of commodity servers coupled with clustering-savvy application servers makes it easy to deploy new applications or add processing capacity to existing applications. But J2EE may have become a victim of its own success. The current approach to capacity planning – provisioning processing resources on a per-application basis, generally using cheap, rack-mounted servers, has created a serious management problem – a proliferation of application host servers in the data center, presenting IT organizations with an increasingly complex management challenge. Data center staff must provision and administer hundreds of servers cost-effectively without compromising system availability, utilization, and service levels.

Network attached processing is a new and innovative approach to data-center management, delivering massive compute power to business applications while reducing the management complexity of today's server farm. Azul is at the forefront of this revolution. By providing traditional host servers with a shared compute pool of mountable ultra-high capacity Azul Compute Appliances, network attached processing provides virtually unlimited processor and memory resources for the Java™ and J2EE platforms.

Working in conjunction with conventional application host servers, the workload is transparently redirected to the compute pool for execution while all interfaces to clients, databases and host operation systems services remain on the application host server. This provides an easy migration path for existing applications, while maintaining the flexibility to execute applications on any appliance in the compute pool. Azul's policy-based Compute Pool Manager™ (CPM) provides flexible and central control of application resources and execution policies. Changes to existing application software, application servers, and other systems are not required.

This paper describes the high-level structure of the Azul virtual machine, the relationship between the Azul VM proxy and the VM engine, and some of the advanced features of the Azul hardware-software combination.

NETWORK ATTACHED PROCESSING

Network attached storage (NAS) technology has successfully enabled companies to reduce costs and enhance flexibility by planning and managing storage capacity on an enterprise-wide basis. Analogously, network attached processing separates the CPU resources from application server hosts, managing a massive pool of compute resources which can serve the requirements of many applications. Figure 1 shows a typical three-tier architecture consisting of a web tier, an application tier, and a data tier; a single enterprise may deploy dozens or hundreds of separate applications

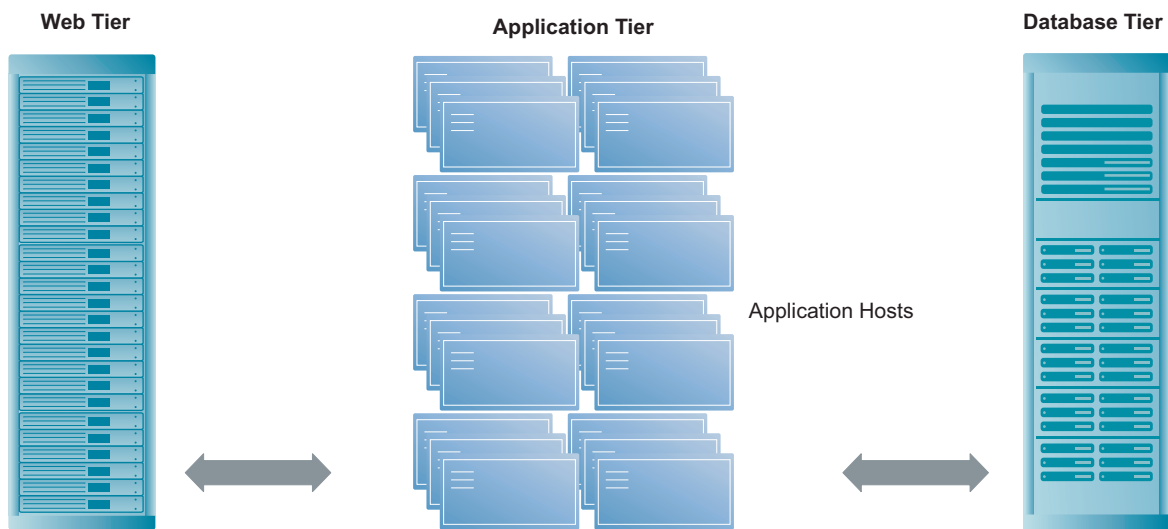


Figure 1 Typical Three Tier Architecture

The web and data tiers are typically treated as “IT infrastructure” and provisioned and managed on an enterprise-wide basis, just like bandwidth, storage, and power. This enables capacity planning on an enterprise-wide basis and lowers costs by increasing resource utilization. The standard practice for the application tier, on the other hand, is to provision capacity on a per-application basis – servers are allocated to a specific application, and cannot be easily shifted to other applications as demand changes. Because application load varies over time, sufficient capacity must be provisioned for each application’s peak load, which often results in application-tier CPU utilization under 10 percent.

Network attached processing enables enterprises to achieve the same consolidation benefits in the application tier that they are already enjoying in the web and data tiers. Figure 2 shows the three-tier architecture with network attached processing technology. The middle tier now consists of two types of systems – traditional host servers (but fewer of them) and a pool of compute appliances. The compute appliances form a massive pool of computing power that can be shared by many applications and managed on an enterprise-wide basis. Azul compute appliances are designed for continuous data center operation, sporting redundant power supplies and network controllers and the ability to route processing around CPU and memory failures for high availability.

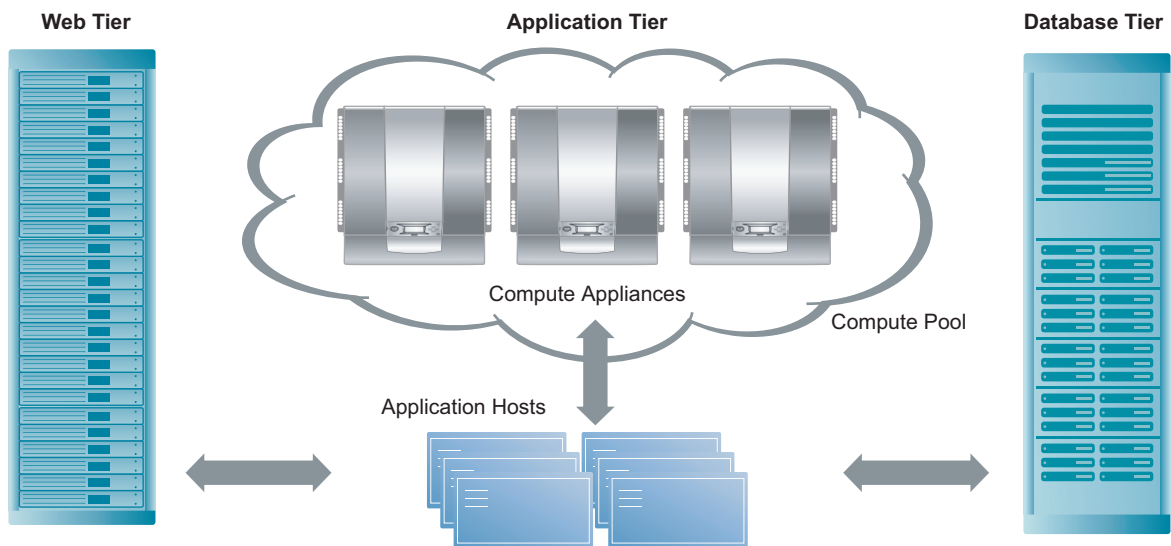


Figure 2 Three Tier Architecture with Network Attached Processing

With network-attached processing, host servers transparently mount virtualized CPU and memory resources, and execute applications on virtual servers within the compute appliance. The Compute Pool Manager (CPM) application provides monitoring of resource utilization across the pool and centralized control of execution policies, allocating CPU and memory resources to applications based on flexible policies. CPU and memory resources can be provisioned and reallocated without any application-level reconfiguration. By provisioning and managing processing resources on an enterprise-wide basis, utilization is improved and management overhead is reduced.

Migrating to a network-attached processing configuration is painless; the only change to host servers is to swap out the existing JVM for the Azul VM proxy. The application software, including the J2EE container and application classes, remains installed and configured on the host server, and the host server accepts request from clients and accesses other middle- or data-tier services just as if the application were executing locally.

THE AZUL VIRTUAL MACHINE PROXY

To tap the computing power of a compute appliance, the JVM on the host server is replaced with the Azul Virtual Machine proxy, which delegates the computational workload to a virtual server running in the compute pool. The VM proxy works in conjunction with the VM engine running on the compute appliance; the application bytecodes are executed on the compute appliance by the VM engine, with socket I/O, file I/O and native code execution delegated back to the VM proxy for execution on the host server. From the perspective of the host server or other hosts, the application is still executing locally because all interaction with clients, databases, and host operating system services is performed through the host server. Because the compute pool offers much more processing power and memory than the host server, each host server can service a much higher volume of requests.

Application Startup

To execute an application on a compute appliance, the application is started on the host server using the VM proxy. The first thing the VM proxy does is contact the Compute Pool Manager, to request access to the pool. The CPM consults its policy database to identify a target compute appliance, which the VM proxy then connects to directly. This startup sequence allows placement to be controlled dynamically by configurable policy parameters such as redundancy requirements and resource availability guarantees, and allows appliances to be added or removed from the compute pool without reconfiguring applications. Figure 3 illustrates the interactions between the host server, CPM, and compute appliance during application startup.

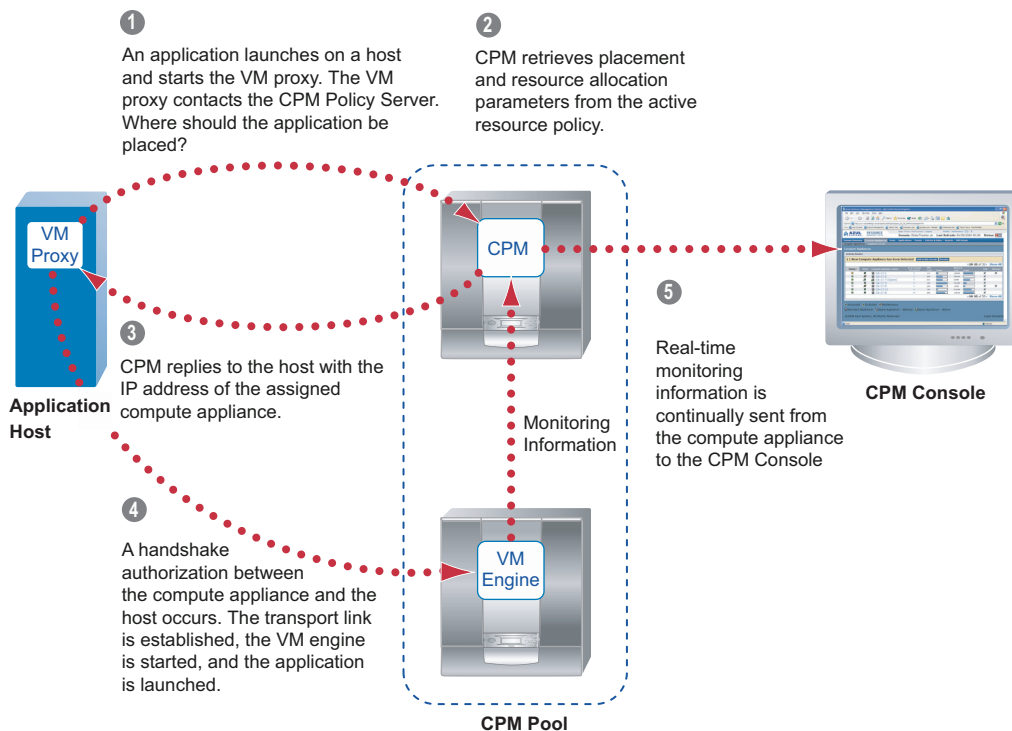


Figure 3 Startup Sequence for Executing an Application on a Compute Appliance

Once the VM proxy and the compute appliance are communicating directly, the proxy uploads the JVM executable for the VM engine to the appliance. By having the client supply the VM engine binary, different applications running within a single appliance can use different JVM versions. The Azul VM engine leverages the hardware capabilities of the compute appliance to add features such as pauseless garbage collection and optimistic thread concurrency, and users can be confident of stability and compatibility because the Azul VM engine is based on Sun Microsystems™ Java HotSpot™ technology and passes the tens of thousands of compatibility and compliance tests required for Sun certification. The operating systems supported as host servers are listed in Table 1. Applications initiated by different host server platforms can execute simultaneously on a single compute appliance, allowing heterogeneous environments to access the compute pool without restriction.

Table 1 Supported Combinations of Hardware, Operating System, and Java Platforms

Application Host Server Hardware	Operating System	Java Software Environment
Sun™ SPARC™ Environment	Solaris™ 8, 9, and 10	BEA® WebLogic® 8.1
		IBM® WebSphere® 5.1 and 6.0
		JBoss™ Application Server 3.2 and 4.0
		Native J2SE
		Caucho® Resin® 2 and 3
		Apache® Tomcat® 4 and 5
Intel® 32-bit Architecture	RedHat™ Linux™ AS/ES 2.1 (Kernel 2.4.9)	BEA WebLogic 8.1
		IBM WebSphere 5.1 and 6.0
		JBoss Application Server 3.2 and 4.0
		Native J2SE
		Caucho Resin 2 and 3
		Apache Tomcat 4 and 5
	RedHat Linux AS/ES 3.0 (Kernel 2.4.21)	BEA WebLogic 8.1
		IBM WebSphere 5.1 and 6.0
		JBoss Application Server 3.2 and 4.0
		Native J2SE
		Caucho Resin 2 and 3
		Apache Tomcat 4 and 5
	SuSe™ Linux 9.x (Kernel 2.6)	BEA WebLogic 8.1
		IBM WebSphere 5.1 and 6.0
		JBoss Application Server 3.2 and 4.0
		Native J2SE
		Caucho Resin 2 and 3
		Apache Tomcat 4 and 5

File I/O and socket I/O are directed to the VM proxy for execution on the host server. From the perspective of the rest of the network, all I/O is performed by the host server, and the application appears to be executing there. The existence of the compute appliance and segmented VM is entirely transparent to the rest of the network. Migrating to a network attached processing environment therefore does not require any changes to existing firewall, security, or other system configurations.

When the VM engine starts up, it loads the main class, core library classes, and other classes required by the application, as well as other resources that are loaded through the class loader such as configuration files. Class loading is done via normal file and network based I/O, which is transparently performed through the VM proxy. Each time a class loader requests a specific JAR or class file, the VM engine accesses the file through the VM proxy.

Once startup is complete, the application is ready to accept requests from clients; clients interact with the application via network connections directed to the host server. The VM proxy transparently forwards these connections to the VM engine, where the application processes the client request. Responses from the application are sent via the VM proxy, which forwards the connection back to the client. When the application initiates an outbound connection to a database server, EJB container, or web service, the VM engine similarly forwards traffic through the VM proxy. From the perspective of network services, the connection is initiated from the host server.

GLOSSARY OF TERMS

Azul Virtual Machine, Engine, and Proxy: The *Azul Virtual Machine* separates a virtual machine into two components; a virtual machine **proxy** and a virtual machine **engine**. The proxy and engine cooperate together to deliver all the services of a virtual machine. The VM proxy runs on the application host in place of a conventional virtual machine. The VM proxy communicates with the operating system on the application host, and with external systems such as the web and database servers. The VM engine runs on the Azul compute appliance and offloads workload from the proxy onto the compute appliance. An engine is started automatically by each proxy, and there is always one engine for each proxy running.

Compute appliance: The name for the type of specialized server devices developed by Azul Systems. Compute appliances provide enormous compute capacity very cost effectively.

Compute pool: A group of Azul compute appliances within the compute pool domain. An appliance can be a member of only one compute pool. Administrators use the CPM Console to define compute pools.

Compute Pool Manager: The Compute Pool Manager (CPM) is a distributed management system that manages Azul compute appliances and the application workload running on those appliances. CPM provides policy-driven management of application workloads, a web-based user interface, event logging, status and alerting for compute appliances and applications,

Bandwidth and latency

It may initially appear that relaying socket connections through the host server would increase bandwidth costs. Perhaps surprisingly, the additional bandwidth required for connection proxying has little effect on application performance in practice. In the worst case, a connection forwarding implementation that simply relayed the raw IP packets would use exactly twice the bandwidth as the equivalent application running directly on the host server. But efficiencies in the proxy-engine protocol that multiplexes data from multiple connections make the true bandwidth cost less than this – if there are many connections proxied through the host server, as there usually will be, data from multiple connections can be combined into a single packet. As packet header overhead often represents a significant fraction of network bandwidth consumption, the larger payload size reduces the packet header overhead, driving the bandwidth multiplier below two. And, because the traffic between the host server and the compute appliance can travel over a private network segment, overall traffic levels on segments used for communication between the host server and clients or services need not increase at all.

The additional bandwidth seen by the host server is unlikely to affect application throughput, as J2EE applications tend to be compute-bound, not network-bound. Most J2EE applications use only a few percent of their network interface capacity, so doubling the level of network traffic at the host server will not constrain application performance at all. Because executing the application on the compute appliance can dramatically increase the capacity of a single host server, throughput for a single host server may eventually be limited by bandwidth – thought at a significantly higher processing volume (see the white paper *Effects of Network Attached Processing: Application Response Time and Network Traffic Levels*)

NATIVE CODE

Java classes can have native methods, whose code is contained in shared libraries instead of Java class files. Since native code is specific to the executable format of the host server, it must execute on the host server via a remote call from the engine to the proxy. But this is only true for native code that is part of the application. All of the native code that is part of the JVM or class library is incorporated into the VM engine, so native code that is not part of the application executes natively on the compute appliance without any network traffic or other remote execution penalty. As very few Java applications use native code that is not part of the class libraries, most applications will not incur any cost at all due to execution of native code.

Relaying client connections through the proxy does add some additional latency, similar to that of an additional network hop, which can make some blocking I/O operations take slightly longer to complete. In percentage terms, however, the effect is likely to be smaller than initially surmised, because of the nature of typical outbound connections. J2EE applications only tend to initiate outbound connections when they access “far away” data such as a database or web service. The latency in servicing a database query is likely to dwarf the incremental latency due to the additional network hop, so the degree to which this extra hop increases perceived response time is negligible.

The other major source of I/O in a typical server application is incoming requests from clients. If every byte read from a client socket required a round-trip to the proxy, this would indeed be a problem. But the proxy and engine cooperate to ensure that this does not happen. Data received from clients is streamed to the compute appliance, where the VM engine buffers it until it is needed by the application. If the data is available, socket reads will complete without any additional latency; if the data is not yet available, the limiting factor is going to be the rate at which the client can send data, not the forwarding overhead. When sending response data back to clients, the additional latency due to the extra network hop is again likely to be dominated by request service time, just as was the case with an outbound connection to a remote resource.

ADVANCED SCALABILITY FEATURES OF THE AZUL PLATFORM

The Azul compute appliance offers far greater computing power than typical host servers – hundreds of processors and hundreds of gigabytes of memory. But simply cramming lots of CPUs and memory into a box doesn’t necessarily mean that applications can scale to exploit it. A combination of hardware features in the Vega processor (see sidebar) and software features of the Azul VM allow enterprises to leverage this massive pool of computing power without modifying their applications. The transparent nature of the Azul VM allows this benefit to be realized without changing the host operating system, replacing or re-installing servers, or porting applications to a new environment.

On traditional processor architectures, the limiting factor for how many CPUs or how much memory can be effectively utilized by a Java application has not been the availability of hardware, but the ability of the garbage collector to keep up with many concurrently executing application threads without incurring long collection pauses. Transaction processing applications often have “soft real-time” response-time requirements, requiring that requests complete or are at least acknowledged within a certain amount of time. If the garbage collector induces a “stop the world” pause that exceeds the maximum acceptable response-time, transactions in progress or queued for execution at the time of the pause will fail to meet these response-time requirements.

Generational garbage collection and the “mostly concurrent” collection algorithm employed by most current JVMs can reduce the frequency of long pauses, but with enough CPUs the application will eventually outrun the collector and a long pause will be required. The Azul platform sidesteps this problem by providing pauseless garbage collection, enabled by a combination of hardware features of the Vega processor and software features of the Azul VM. This allows Java applications to break out of the “comfort zone” of sub-gigabyte heaps and a handful of processors, and effectively utilize dozens or hundreds of processors and heaps as large as 96GB.

Coarse-grained resource locking within applications is another factor that can limit throughput despite the availability of additional CPUs. When a lock protects a resource that is frequently read but rarely modified, only one thread can access the data at once even when multiple reading threads could have shared it safely. While this could be addressed by modifying the application to use finer-grained locking or a multiple-reader, single-writer locking discipline, this is difficult and error prone. Instead, the Azul platform provides optimistic thread concurrency, which can detect when multiple threads are contending for a lock but not contending for any data, and safely allow a greater degree of concurrent access.

PAUSELESS GARBAGE COLLECTION

Garbage collection technology has improved dramatically in recent years, consuming fewer CPU resources and offering shorter pause times than older collectors running on the same hardware. But hardware technology has improved too – systems have more processors and heaps have grown larger. At the scale of existing commodity server hardware, some companies have chosen not to adopt Java technology because of unpredictable pause times or chosen to deploy applications on artificially small JVMs. At the scale of the Azul Compute Appliance, with 384 processor cores and 256 gigabytes of memory, existing garbage collections algorithms would exhibit totally unacceptable pauses. To support a system with hundreds of processors (and therefore hundreds of application threads allocating memory and modifying references at the same time) required a new approach to garbage collection. Through a combination of hardware and software, Azul developed a concurrent, parallel, relocating garbage collector that supports true pauseless operation to deliver predictable response time even for JVMs with hundreds of processors and heaps as large as 96 gigabytes. The result is the ability to scale Java applications to an entirely new level.

The existing concurrent collectors, such as those provided by HotSpot 1.4.1 and later, are in fact only “mostly concurrent” – while some of the garbage collection cycle can run concurrently with the application, there are still potentially long stop-the-world pauses whose duration increases with processor count. For HotSpot’s concurrent mark-sweep collector, the marking cycle is divided into phases – a short “initial mark” phase, where application threads are stopped so the root set can be identified; a “concurrent mark” phase, where a single collector thread runs concurrently with application threads; and a “remark” phase, where application threads are again stopped so that the collector can catch up with changes made by the application threads during the concurrent mark phase. The problem with the “mostly concurrent” approach is that the duration of the remark phase, during which all application threads are again paused, increases with the number of object references modified by the application threads during the concurrent mark phase. With a handful of processors and heaps of a few hundred megabytes, the application threads will probably have not made too much extra work for the collector during the concurrent mark phase, but with hundreds of processors and multi-gigabyte heaps, it is likely that the collector thread will be overwhelmed with changes and will have to pause the application for a long time. The “mostly concurrent” garbage collection algorithm simply does not scale to larger virtual machine configurations.

Concurrent marking and relocation

In the Azul pauseless collection algorithm, the garbage collector threads run concurrently with the application threads, marking objects known to be live. The runtime environment can monitor the relationship between the reclamation rate of the collector threads and the allocation rate of the application threads; if these are out of balance, more collector threads are added, ensuring that application threads can make progress without running out of memory. The application threads cooperate with the garbage collector by assisting in the marking process when they come across an object that the collector has not yet marked. This is done with hardware assistance; the memory subsystem provides some support for the garbage collector, allowing it to distinguish between marked and unmarked objects when the collector is in the mark phase. If an application thread uses an object that is not yet marked, it will trap to a fast user-mode trap handler that marks the object and perform a small amount of garbage collection processing. Because the object is now marked, subsequent use of the same reference by the application will not trigger another trap in the same garbage collection cycle. The hardware support for identifying which objects have been marked also simplifies the collection algorithm by eliminating a multitude of race conditions between application and garbage collector threads.

Another problem in concurrent garbage collection algorithms is object relocation. Without object relocation, the heap can become fragmented; by relocating objects at garbage collection time, heap allocation can be made far more efficient. If a memory page is sparsely populated with

objects at the end of a garbage collection cycle, the few remaining live objects in that page can be relocated to another area of memory. In a stop-the-world tracing collector, it is easy to relocate objects, as references can be updated as the heap is traced free of interference from application threads. But in a concurrent relocating collector, an efficient mechanism is needed for moving objects even if they are in use by other threads. Again, hardware features of the Vega processor help achieve this goal. When a page containing live objects is reclaimed, the objects are copied elsewhere and their forwarding addresses recorded, virtual memory protection is used to mark the page as relocated, and the physical memory is reclaimed. If a thread still holds a reference to the old location, when that reference is used a trap is triggered which resolves the reference and updates it so that the trap will not be triggered again for that reference during this collection cycle. During the next garbage collection cycle, remaining references to relocated objects can be updated so that the virtual memory space occupied by the relocated page can then be reclaimed.

The result is a concurrent, parallel, relocating, compacting garbage collector which eliminates stop-the-world pauses and delivers unprecedented scalability and response-time predictability. For more information on pauseless garbage collection, see the white paper [Improving Application Scalability and Predictability with Pauseless Garbage Collection](#).

THE VEGA PROCESSOR

Powering the Azul Compute Appliance is the Vega processor (actually, many Vega processors), a new general purpose processor designed for running virtual machines in highly concurrent environments. The Vega chip includes features not found in conventional processors, enabling a variety of optimizations that would otherwise be impossible. With support for features such as read and write barriers that help optimized garbage collection and object relocation, speculative locking to enable safe concurrent execution of code that would otherwise be serialized, and an instruction set designed for the needs of virtual machines, the Vega processor is designed to provide consistently high throughput to Java applications.

The Vega chip is a 64-bit RISC processor, with 24 independent processor cores per chip and multi-chip coherency technology. An appliance can support up to 16 chips for a total of 384 processor cores in a symmetric multiprocessor (SMP) architecture. Memory access is uniform across all cores through a passive, non-blocking interconnect mesh, and MOESI cache coherency. A fast uncontended CAS operation reduces the need for JVMs to rely on pipeline-stalling memory barrier instructions for common runtime operations such as lock management.

OPTIMISTIC THREAD CONCURRENCY

Amdahl's law describes the performance of a system as more processors are added. In the limit, performance is limited by the portion of the workload that cannot be parallelized. As shown in Figure 2, even if only 1% of the code must be executed serially, an application cannot be sped up by more than factor of 100, no matter how many processors it has available to it. On the other hand, code which can minimize serial execution can achieve dramatic speedups.

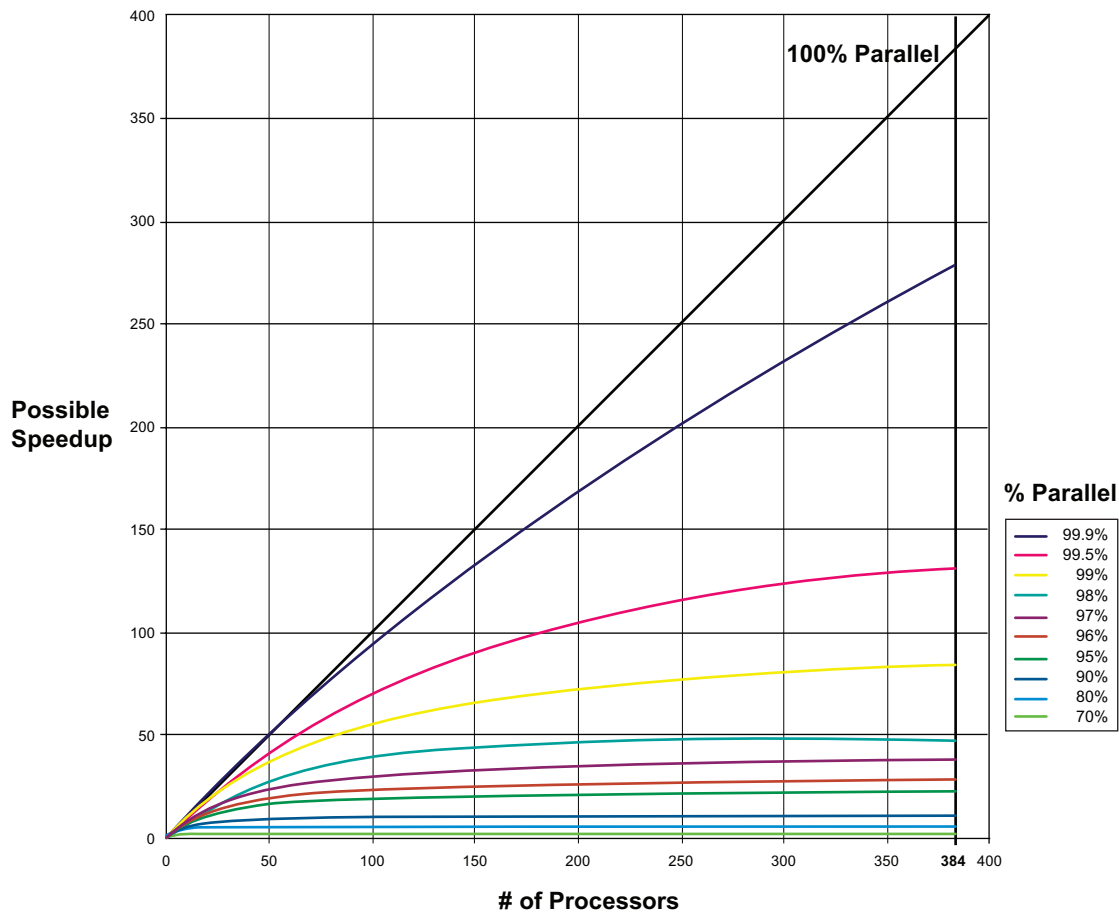


Figure 2 Amdahl's Law Effect of Throughput: Possible Speedup Affected by Serialization Percentage and Number of Processors

The Java language relies on synchronization for coordinating access to shared state. Synchronization enforces a mutual-exclusion semaphore (mutex) to guarantee exclusive access to shared data. While this mechanism can provide thread-safety, this safety comes at a cost: operations which access shared state must be serialized. In practice, many Java classes use relatively coarse-grained locks to coordinate access to shared state. On a system with a handful of processors, coarse-grained exclusive resource locks are not necessarily a serious scalability bottleneck. But applications which exhibit acceptable throughput on a four-way system may encounter limits in scalability as the processor count increases.

On a system with a handful of processors, and therefore only a handful of concurrently executing threads, coarse-grained exclusive resource locks are not necessarily a serious scalability bottleneck. But applications which exhibit acceptable throughput on a two- or four-way system may encounter limits in scalability as processor count increases. As the Azul platform supports hundreds of processors, code which uses coarse-grained, exclusive resource locks to ensure thread-safety appears to be on a collision course with Amdahl's law. But a feature of the Azul architecture, optimistic thread concurrency (OTC), enables the Azul VM to identify code paths that can safely be executed concurrently even when using exclusive locks, so as to eliminate the serialization bottlenecks imposed by the overly coarse-grained locking used by many Java classes.

Just as with pauseless garbage collection, OTC is accomplished by a combination of hardware and JVM support. The Vega processor supports a "speculative locking" memory access mode where multiple threads can be granted the same lock simultaneously and the hardware will detect if they actually contend for the same data. If there is no data contention, throughput has been improved at no cost to safety by removing unnecessary serialization; if there is data contention, the hardware detects this and alerts the JVM, which informs the hardware to "roll back" changes made since the lock was speculatively acquired. In this way, OTC implements a form of software transactional memory, using synchronized blocks to demarcate memory transactions.

This technique is called optimistic because it amounts to assuming that conflicts will be infrequent and obtaining forgiveness later (rolling back the transaction) in the rare case that a conflict occurs, instead of obtaining permission ahead of time (acquiring all locks exclusively before proceeding.) The result is that existing code which may have serious scalability bottlenecks due to coarse-grained locking can be executed safely with a much higher degree of concurrency without software changes. For more information on optimistic thread concurrency, see the white paper [Optimistic Concurrency – innovative locking technology from Azul Systems](#).

64-bit heaps on 32-bit hosts

The performance of traditional garbage collectors may impose a limit on the heap size for a single JVM; another limit on heap size is the address space of 32-bit memory architectures. Because the operating system and JVM share the address space with the application, it is difficult to run Java applications with heaps larger than 3GB on 32-bit platforms.

The Azul platform enables applications to use much larger heaps – up to 96GB – even when the host server is a 32-bit machine. Because the application executes on the compute appliance, which has a 64-bit address space, application heaps are not constrained by the address space of the application server host, and applications can exploit the 64-bit address space without replacing the server host hardware.

CONCLUSION

Working hand-in-hand with the Vega processor, the Azul VM enables Java applications to move beyond the limitation of sub-gigabyte heaps and a handful of processors to effectively utilize dozens or hundreds of processors and heaps as large as 96G, enabling unprecedented scalability and response-time predictability for Java applications.

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 February 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, Azul, the Azul arch logo, Compute Pool Manager, and Vega are trademarks of Azul Systems Inc. in the United States and other countries. BEA and WebLogic are registered trademarks of BEA Systems, Inc in the United States and other countries. IBM and WebSphere are trademarks of International Business Machines Corporation in the United States and other countries. JBoss is a registered trademark of JBoss Inc. Intel and IA-32 are registered trademarks of Intel Corporation in the United States and other countries. Linux is a registered trademark of Linus Torvalds. RedHat is the property of Red Hat, Inc. 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. SPARC is a trademark of SPARC International, Inc. in the United States and other countries. Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or 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.

