Project Kittyhawk: Building a Global-Scale Computer

Blue Gene/P as a Generic Computing Platform

Jonathan Appavoo Volkmar Uhlig Amos Waterland
IBM T.J. Watson Research Center, Yorktown Heights, NY

Abstract

This paper describes Project Kittyhawk, an undertaking at IBM Research to explore the construction of a next-generation platform capable of hosting many simultaneous web-scale workloads. We hypothesize that for a large class of web-scale workloads the Blue Gene/P platform is an order of magnitude more efficient to purchase and operate than the commodity clusters in use today. Driven by scientific computing demands the Blue Gene designers pursued an aggressive system-on-a-chip methodology that led to a scalable platform composed of air-cooled racks. Each rack contains more than a thousand independent computers with high-speed interconnects inside and between racks.

We postulate that the same demands of efficiency and density apply to web-scale platforms. This project aims to develop the system software to enable Blue Gene/P as a generic platform capable of being used by heterogeneous workloads. We describe our firmware and operating system work to provide Blue Gene/P with generic system software, one of the results of which is the ability to run thousands of heterogeneous Linux instances connected by TCP/IP networks over the high-speed internal interconnects.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; C.5.1 [**Computer System Implementation**]: Large and Medium ("Mainframe")—*Super (very large) computers*

General Terms

Design, Reliability, Performance, Management

1. INTRODUCTION

Project Kittyhawk's goal is to explore the construction and implications of a global-scale shared computer capable of hosting the entire Internet as an application. This research effort is in an early stage, so this paper describes our conjectures and our ongoing work rather than a completed set of results.

The explosive growth of the Internet is creating a demand for inexpensive online computing capacity. This demand is currently fulfilled by a variety of solutions ranging from off-the-shelf personal computers connected to the Internet via broadband to large clusters of servers distributed across multiple data centers. The general trend however is to consolidate many machines in centralized data centers to minimize cost by leveraging economies of scale. This consolidation trend includes building of data centers near hydroelectric power plants [8], colocating physical machines in large data centers, over-committing physical hardware using virtualization, and software-as-a-service.

At present, almost all of the companies operating at web-scale are using clusters of commodity computers, an approach that we postulate is akin to building a power plant from a collection of portable generators. That is, commodity computers were never designed to be efficient at scale, so while each server seems like a low-price part in isolation, the cluster in aggregate is expensive to purchase, power and cool in addition to being failure-prone. Despite the inexpensive network interface cards in commodity computers, the cost to network them does not scale linearly with the number of computers. The switching infrastructure required to support large clusters of computers is not a commodity component, and the cost of high-end switches does not scale linearly with the number of ports. Because of the power and cooling properties of commodity computers many datacenter operators must leave significant floor space unused to fit within the datacenter power budget, which then requires the significant investment of building additional datacenters.

Many web-scale companies start in a graduate lab or a garage [18], which limits their options to the incremental purchase of commodity computers even though they can recognize the drawbacks listed above and the value of investing in the construction of an integrated, efficient platform designed for the scale that they hope to reach. Once these companies reach a certain scale they find themselves in a double bind. They can recognize that their commodity clusters are inefficient, but they have a significant investment in their existing infrastructure and do not have the in-house expertise for the large research and development investment required to design a more efficient platform.

Companies such as IBM have invested years in gaining experience in the design and implementation of large-scale integrated computer systems built for organizations such as national laboratories and the aerospace industry. As the demands of these customers for scale increased, IBM was forced to find a design point in our Blue Gene supercomputer technology that allowed dense packaging of commodity processors with highly specialized interconnects and cooling components. Our aim is to determine whether IBM's deep institutional knowledge, experience and capability in super-computing—specifically Blue Gene—coupled with this design point can be leveraged to address the problems facing web-scale computing. The system software work described in this paper is aimed at making the incremental purchasing

of Blue Gene capacity as attractive as the current practice of incremental purchasing of commodity computers.

We postulate that efficient, balanced machines with high-performance internal networks such as Blue Gene are not only significantly better choices for web-scale companies but can form the building blocks of one global-scale shared computer. Such a computer would be capable of hosting not only individual web-scale workloads but the entire Internet.

This paper is structured as follows: Section 2 describes the overall vision of the project and Section 3 discusses our motivating principles regarding large-scale hardware and software. Section 4 describes the Blue Gene/P platform and the firmware and operating system work we have done for it. Section 5 describes our first experiences running the platform as a generic heterogeneous computer and Section 6 concludes.

2. VISION

We assert that the keys to success of building a global computational infrastructure (besides the obvious economic ones) are to embrace diversity, ingenuity, cooperation and competition. The intention is not to build a single static system from which a single corporation provides and controls all the value from it. Rather the intention is to create a shared resource in which global benefit can be derived from the advantages of consolidation, while enabling individuals and organizations to provide sustainable, unique, and potentially both cooperative and competitive offerings.

As such the vision we have for our system is very much in the tradition of systems design. We are focusing on providing a basic set of primitives which are both sufficiently primitive to encourage building arbitrary solutions while being high-level enough to help constrain the complexity and burden of implementation. With this in mind we propose a system which is designed to allow others to build computational services for many aspects of computation including: systems management, network configuration, optimization, service construction, migration, over-commitment, and reliability.

We also assert it is critical to provide a model for the system to be incrementally deployed and sustained. A global system has more than just technical impacts but also economic and social impacts. These topics will not be covered in this paper but are considered in the project. In general, our approach is to enable building blocks that allow others to explore and provide creative solutions.

From a technical perspective our goal is to isolate the benefits of hardware from those of software and allow as much of the software value to be provisioned by a diverse population of parties. The system is predicated on supporting current models and practices while evolving them by enabling arbitrary forms of virtualization (hardware, software, and higher-level services) by diverse sets of providers. Our aim is to minimize the software that the system enforces in order to allow others to add arbitrary services. The system, however, is designed to ease the software/service providers' burden by enabling primitives and examples for management, billing and development. From a hardware perspective a consolidated platform has the opportunity to redefine what is considered commodity. The design of the components and structure of the machine can be optimized not based on third party packaging and pricing models but rather to meet the demand of efficient cost-effective scalable computation. Upgrade and optimization can be driven by real demand rather than artificially created upgrade cycles due to business models based on server replacement rates. In time new hardware can be introduced based on proven cost benefit analysis.

The abstract vision is an underlying system on which base providers can obtain hardware-backed processing, memory and communication resources. The resources are provided in the form of fully documented raw nodes composed of processors, memory, and communication interfaces. The use of the resources will be metered, reported, and billed on an individual resource basis to the owner[1].

In our consolidated machine model, the owner of resources can construct resource and communication domains to restrict which nodes can directly communicate with each other. The owner is provided with the following primitives for controlling nodes:

- Secure Reset/Power cycling of individual or groups of nodes,
- Secure communication to a simple boat loader (which can be replaced) running on the nodes at startup,
- Secure debugging facilities similar to a JTAG interface [12],
- Secure management interface to reconfigure the network topology and routing configuration,
- Defined hardware failure model,
- Example software for using nodes and the communication facilities (including device drivers and a complete open OS and user software environment).

The owner is free to use the processing resources as she wishes such as reselling, over-committing, and offering management.

In order to build scalable services, we provide primitives for simultaneously communicating with groups of nodes, such as for boot-strapping a cluster with a common software stack or debugging multiple nodes at once.

In the rest of this paper we will discuss our prototyping of this vision on Blue Gene/P. Availability and support of standard software, interfaces and protocols is paramount; we provide an example Linux-based kernel and system stack. The machine does not have all hardware aspects our vision argues for and in part our goal is to explore and prototype these new hardware requirements.

3. SCALABILITY AND SUSTAINABILITY

We assert that a global-scale system requires scalable hardware and a sustainable model for software provisioning. In this section we discuss cluster and SMP designs and suggest that a hybrid machine is more appropriate for a global-scale system. We then discuss our principles for provisioning large-scale software services on such a system.

[1]The units of metering and billing models are beyond the scope of this paper.

3.1 Structure of Scalable Hardware

When we think of today's large scale computers the two main-stream structures we consider are shared memory multiprocessors, tightly coupled systems, and clusters, loosely coupled systems. One might easily conclude that, given the structure of the current Internet, a global scale computer's structure would inherently be that of a loosely coupled system. We contend however, that neither today's clusters or shared memory multiprocessor systems are ideal candidates for a global scale computer.

Rather we propose that a hybrid is a more appropriate structure to seed the evolution of a system. In prior work we have studied the construction of scalable Non-Uniform Memory Access (NUMA) SMP systems and software [1,10]. Such NUMA systems, including related Cache Only Memory Access (COMA) systems, have a cluster like architecture and are designed to scale from a few processors to hundreds of processors in a modular fashion. They are built from nodes, containing a small number of processors and memory, connected by a scalable inter-node interconnect. For example, in the case of NUMAchine, the interconnect is a hierarchy of high-speed bit-parallel rings. Typically these systems were constructed to provide a single-system abstraction via hardware supported shared memory.

The interconnect network was generally not directly visible to the software, rather it was utilized as a memory bus. Despite the shared memory support we found that when designing and implementing the software for such systems it was critical to exploit locality by utilizing distributed systems techniques [1]. Given the underlying shared memory support of these multiprocessors, systems software (operating systems and hypervisors) are used to provide isolation. The software allows multiple computations to be executed in parallel and multiple users to securely share a single system. There is, however, an implicit acceptance of a shared software layer which is installed and owned by a single party which is responsible for resource management.

In contrast, loosely coupled clusters do not impose a hardware enforced shared memory model between nodes. The physical communications topology and policies, however, are hardware enforced, typically by wiring and switching hardware. Access and configuration of a cluster's wiring and switching hardware is typically controlled by a single party. Users of a cluster must request specific wiring and switch configuration, in order to obtain the physical communication topology and policies they require. Alternatively, the cluster can be run in an open configuration, in which all nodes of the cluster can communicate with each other, and users employ per-node firewall and/or encryption software for isolation. In this scenario users must accepting the risk of both intentional and unintentional communications to their nodes and the associated costs and side effects, as is typified by an intentional denial-of-service attack or unintentional errors in software or hardware.

Given the trend to resurrect machine level virtualization [9] one might assume that the underlying structure of the hardware need not impact the virtual machine and virtual network configuration that independent users desire. It is worth remembering however, that virtualization cannot create something from nothing. It is possible to construct a virtual SMP of 16 processors from four 4 processor nodes of a cluster [2,14] but in practice the network may not be appropriate to act as a memory interconnect. In contrast using a global shared hypervisor to carve off 16 independent uniprocessor virtual machines connected to isolated networks from a large SMP may not provide the desired isolation given the underlying hardware sharing.

We claim that a hybrid multiprocessor designed to support global web-scale numbers of individual owners should have the following properties:

- large scale consolidated manufacturing processes,
- efficient packaging, powering, cooling and installation, exploiting large homogeneously structured units of hardware,
- simple and minimalist node architecture,
- scalable NUMA-like interconnect bus for raw general purpose communications,
- configurable, hardware enforced, communications domains, and
- scalable control and management hardware.

A simple abstract metaphor is to imagine taking a rack of 1U servers and turning each server into a node which looks like an enlarged SIMM module, with memory chips and a system-on-a-chip processor which incorporates additional bus controllers. Then packaging these into rack sized quantities which connect the nodes to a control network and a general purpose interconnect. In many respects, Blue Gene/P provides a realization of this hybrid model. Section 4 describes in further detail the Blue Gene/P architecture.

3.2 Sustainable Software Provisioning

We assert that the following principles are necessary for the sustainable provisioning and administration of a large-scale collection of computers. Our use of the word "sustainable" refers to a property we desire in the continuing care and maintenance of large-scale software services deployed on thousands of nodes.

Statelessness: The software running on each node should not store permanent state locally. Local administrative actions should have no permanent effects and it should not be possible to permanently damage by mistake or malice the software on a node. A power cycle should restore the node to a known good state, and in a transitive manner a system-wide reboot should restore the system to a known good state. "Stateless" does not imply the lack of state altogether, it implies a careful management and separation of local transient state from permanent state.

Locality: The software running on each node should in normal operation consult no entities other than itself. For example, nodes should not be contending on a central server to obtain tokens for shared read-only files. This does not imply a lack of communication entirely, it implies a careful separation of files that are truly used for communication from files that are common but read-only.

Commonality: System files should be identical across nodes, and administrative actions should be done only once

but reflected across sets of nodes in an atomic, time-bounded and scheduled manner. The reflection of the shared state should be an O(1) operation.

Auditability: Administrative actions, such as the edit of a configuration file or the install of a new application, should leave an audit trail similar to the version control history of software projects. Administrative actions should be transactional and should be able to be rolled back.

A collection of machines provisioned in a manner consistent with the above principles contrasts with a collection of machines installed in the traditional ad-hoc server model. In the latter, individual administrative actions over time produce increasing entropy in which the state of each machine diverges permanently from that of its peers.

Many software objects that are currently called “appliances” are not stateless and auditable. It is true that they do not require configuration, but their runtime state is capable of being damaged with no simple push-of-a-button reset capability. For example, one can download a canned web server virtual appliance and then immediately lose the property of statelessness by storing the main copy of one’s website in it. At this point one must speak in the singular of the web server as opposed to an arbitrary collection of stateless machines of which any and all in parallel may serve as the web server.

4. BLUE GENE PROTOTYPE

The Blue Gene super-computer has highly desirable properties in the areas of density, cooling costs, and performance per dollar for a company wishing to provide massive compute capacity on demand. In this section we describe the relevant aspects of the Blue Gene platform and our software layer as provided to the resource owners.

4.1 Node Model

A Blue Gene/P node is composed of four 850 MHz cache-coherent PowerPC cores in a system-on-a-chip with DRAM and interconnect controllers. The nodes are grouped in units of 32 on a node card via the interconnects and the node cards are grouped in units of 16 onto a midplane. There are 2 midplanes in a rack providing a total of 1024 nodes and, in its current configuration, a total of 2TB of RAM. Multiple racks can be joined together to construct an installation. The theoretical hardware limit to the number of racks is 16384. This results in the maximum size in nodes of an installation being 16.7 million and in cores 67.1 million with 32 Petabytes of memory. See Figure 1 for a photo of a Blue Gene/P node and node card.

Additionally, each node card can have up to two IO nodes featuring the same basic unit but with an additional 10G Ethernet port. Hence, each rack has a external IO bandwidth of up to 640 Gbit/s; the aggregate IO bandwidth of the maximum installation is 10.4 Petabit/s.

The key fact to note is that the nodes themselves can be viewed for the most part as general purpose computers, with processors, memory and external IO. The one major exception to this is that the cores have been extended with an enhanced floating point unit [11] to enable super-computing workloads.

4.2 Communications and Network models

The interconnects seamlessly cross all the physical boundaries described before, at least with respect to the software, once the machine is configured. There are four relevant networks on Blue Gene/P: (1) a global secure control network, (2) a hierarchical collective network (which provides broadcasts and multicasts), (3) a three-dimensional torus network, and (4) a 10 Gbit Ethernet for external communication.

4.2.1 Dynamic Network Topologies

Blue Gene’s networks provide all-to-all connectivity. Packets flow from node to node and either get deposited in the receive buffers or forwarded to the next neighboring node. The internal networks are all reliable; packets are guaranteed to be delivered.

The hardware allows for electric partitioning of nodes at the granularity of 16, assuming the allocation has at least one IO node in the network. In our typical environment we use one IO node for one or two node cards, thus having a minimal allocation size of 32 or 64 nodes.

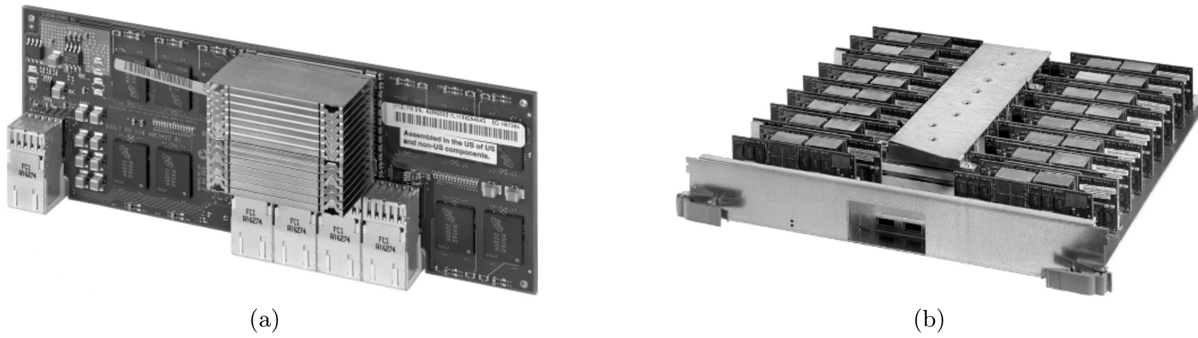
The collective and torus networks form our basic link layer; point-to-point communication is best performed on the more powerful torus while broadcast and multicasts are most efficient on the collective network. The functionality of the torus network is comparable to an Infiniband network including remote-DMA capabilities. The collective network supports up to 16 routes which allow for flexible creation of communication domains within subsets of nodes thereby minimizing the payload on the global network.

At this point we lack the hardware capabilities to restrict communication beyond the electrical partitioning. Instead we use a (trusted) virtualization layer to enforce communication permissions.

4.2.2 Network and Storage

We packetize and multiplex standard protocols such as Ethernet and the console on top of the underlying communication fabrics. We achieve external connectivity by using IO nodes as network routers or bridges which then forward the packets to the external network. However, routing is not limited to external networks; standard network techniques such as firewalling, IP masquerading, and proxying are all applicable between Blue Gene nodes. We therefore can construct isolated and secure subnets as independent trust domains and enforce security properties on the networking layer.

Blue Gene does not contain internal storage and we therefore rely on externally provided disk capacity. From our initial experiments with a high-end server providing NFS we quickly learned that scalability is quite limited: booting a 128 node configuration over NFS is nearly impossible due to constant network timeouts. We are trying to push as much of the storage complexity as possible into Blue Gene and are pursuing two directions. First, we use storage devices which operate on the block level, such as the Coraid’s EtherDrive [3]. The disks can then be accessed by multiple nodes in parallel and coherency mechanisms can be implemented over Blue Gene’s high-performance networks. Second, we try to use the internal memory as storage. This is feasible considering the sheer amount of available and quickly accessible memory capacity of 2 TBytes per rack.



(a) (b)

Figure 1: Blue Gene/P components. (a) compute node with a 4-core PowerPC 450 and 2GB RAM, (b) populated node card with 32 compute nodes and 2 IO nodes (two 10G Ethernet connectors on the front). 32 node cards form a rack.

4.3 Control and Management Model

Blue Gene's original control system was constructed for a very challenging task: booting a single "task" on 64000 nodes within a few minutes. The unit of allocation is a block which contains from 16 up to all nodes in an installation.

For a general purpose environment the existing control system has a primary limitation which is that all nodes run the same software image. While in many cases we may use the same kernel image, we want to be able to customize the boot-strap ramdisk image and command line parameters to the OS kernel. Instead of taking the obvious approach of extending the control system we took a different route which enables a very flexible resource allocation and management scheme.

Blue Gene's control network provides access to all parts of the hardware and is therefore inherently insecure. Exposing it out to many clients requires a sophisticated security infrastructure, scalable management infrastructure, scalable storage for storing boot images, scalable network connectivity etc. We try to push as many of these services as possible into Blue Gene and use Blue Gene resources themselves for management and accounting. This approach inherently scales: every time a new rack comes online a fraction of that rack's resources is used for management.

Our node allocation still happens in blocks, however, we allocate them in large quantities into a standby pool. Each of the nodes is boot-strapped with a small but powerful firmware (see also Section 4.5) which allows interaction with the node over the network. From the standby pool nodes are then allocated and accounted to individual user accounts. Upon each node allocation we return a list of node addresses which are available for use. When the user deallocates a node, the node is reset, the memory gets scrubbed, and the node placed back into the pool ready to be handed out again.[2]

4.4 Reliability and Failure Model

Reliability is a key problem at scale: machines fail. Here, Blue Gene has a significant advantage over commodity platforms. Blue Gene was designed for a reliability target of 7 days mean time between failure for a machine with 72 racks, which is roughly 73,000 nodes and 146TB of memory. Since the machine is built to run one application, a single node failure is considered a machine failure. Hence, individual node reliability is two orders of magnitudes higher than a commodity server.

Blue Gene's control network is also used for failure reporting back to the control system. The control system primarily reports the errors into a central database which is then used to deactivate faulty components and route around errors.

We are extending the existing infrastructure to allow nodes to actively react to hardware failures. Node failures are in many cases non-fatal for the application and recovery is possible. However, node failures which traditionally do not affect a node need to be handled due to the high level of integration. For example, when a node fails which acts as a forwarding node at the physical layer, a network segment may become unreachable. While we can easily deallocate the faulty node from the pool, we must ensure that all necessary nodes still provide networking functionality. Here, the reliability of a single-chip solution is very advantageous. The failure of nodes are often due to failing memory modules. However, each processor chip has 8MB of integrated eDRAM. If more than one RAM chip fails we can usually bring the node back into a state where it still acts as a router, even though normal workloads cannot be run.

The key to handle hardware failures gracefully is a deterministic failure model. For example an operating system may recover from certain RAM failures if the necessary recovery code is located in memory with error correction and higher reliability. Furthermore, networking errors can be compensated via an out-of-band channel through the control network. We make each node explicitly aware of the hardware outages so that the nodes, for example, reconfigure the network links and route around outages.

4.5 Boot Loader

As mentioned before, the Blue Gene control system only allows booting a single image on all nodes of a block. Nodes are completely stateless; there is no flash or ROM in any of the nodes. Thus, even the initial firmware which initializes the hardware is loaded via the control network into each node of the machine.

We take over the node from the firmware via a generic boot loader which is placed on all nodes by the existing

[2]This scheme also fits seamlessly into the primary use of our Blue Gene installation, namely scientific computing. Allocating nodes out of a pre-allocated pool makes this pool appear to be running a scientific application to the other users of the system.

control system. The boot loader, U-Boot [5], offers rich functionality including a scripting language, network booting via NFS and TFTP, and a network console. We extended U-Boot with the necessary device drivers for the Blue Gene hardware and integrated the Lightweight IP stack [6, 7] to provide support for TCP as a reliable network transport.

With this base functionality, individual nodes can be controlled remotely, boot sequences can be scripted, and specialized kernel images can be uploaded on a case-by-case, node-by-node basis. We envisage two models for boot-strapping: a pull and a push model. In the pull model, a node fetches its images from a network location, such as an NFS server.

In the push model, the controlling host pushes the kernel images into the node. The push approach has a number of advantages. First, no public storage is required to bootstrap a node. An owner/user of a node can directly load a kernel and boot image (boot stack) from their client device or they can construct services which automatically acquires a node and pushes the appropriate boot stack to meet the demand that triggered the need for an additional node. Second, since a node does not need to independently connect to a storage server, no security-sensitive authentication credentials need to be transmitted to a node's boot loader. Third, parallel boot-strapping is simplified as a push model is more amenable to the use of multi-cast communication.

We added a simple HTTP server to U-Boot so that individual nodes can be booted via a `PUT` command that pushes the desired boot images and kernel command line. The command line is evaluated via the scripting environment before it gets passed on to the OS kernel thereby allowing per-node customizations. With this simple extension to U-Boot we are able to construct powerful scripted environments and bootstrap large numbers of nodes in a controlled, flexible, and secure way.

4.6 Operating System

At this point we are working on two system stacks, Linux as a general purpose operating system and the L4 hypervisor/microkernel [15]. Linux provides us with a standard UNIX environment and a rich and well-tested software stack. We added device drivers for the key devices in Blue Gene: the collective network, the torus network, the external Ethernet adapter, the interrupt controller, and a bring-up console. Layered upon the internal networks we added support for an Ethernet adapter which supports jumbo packets. Surprisingly, these few drivers made the system available for a huge amount of general purpose workloads.

We are aiming for a simple virtualization solution to experiment with new system software models and hardware extensions. L4 gives us a good lightweight virtualization layer where we can experiment with new resource allocation and sharing models. L4 provides another important feature: Blue Gene's current security model is that of a single application and all nodes are trusted among each other. A security domain spans the size of a block allocation which can be thousands of nodes. A trusted intermediary software layer, such as a hypervisor, enables us to enforce admission control to the hardware. When the hypervisor work is completed we will run the boot-loader as a virtual machine on top of L4 and we can guarantee a root of trust [16].

5. FIRST EXPERIENCES

In this section we describe our first experiences using a super-computer for general purpose workloads. As expected, when using standard practices we experienced problems related to the scalability of core services like NFS, DHCP, and LDAP.

5.1 Software Packaging

A common approach to centrally manage a large number of servers is to use diskless machines that retrieve their system files from a central network file server. In [4] we describe an implementation of this model where the transient per-node state is maintained in local RAM disks mounted over the permanent centralized global state maintained in a read-only export. As an increasing number of nodes using this model access the centralized storage, the network server becomes a bottleneck despite the fact that all accesses are read-only.

In order to address our scalability problems of running large numbers of nodes, we pursued an approach of building small root filesystem images that can be efficiently loaded into RAM during boot. We built a tool that automatically generates from a stock Linux filesystem an image containing only the files necessary to run a given application. We call this packaging a "software appliances" to emphasize that they are a tiny but complete working system.

Table 2 presents the size of the software appliances our tools automatically constructed from a typical Linux root filesystem of around 2 GB. The appliances are generally 5% of the size of the full root filesystem.

Appliance	Size (MB)
Shell	3
SSH Server	10
x86 Emulator	11
Ruby on Rails Website	12
SPECint2006	14
SPECjbb2005	90

Figure 2: Software appliance sizes.

The images are packaged in a format that Linux supports as a pure RAM image that does not require double buffering to a block store. Note that by using just a Linux kernel plus a RAM image we are able to enjoy the benefits of software appliances without requiring virtualization.

This approach allows us to reprovision a block of 512 nodes in about 45 seconds, switching for example from Java workers to Ruby on Rails web servers etc. We built tools to edit and merge images, so that the use of the tool for image capture is only necessary when determining the filesystem requirements of a new application.

The principles that make this approach successful are discussed in Section 3.2. Specifically, since each appliance is has no permanent state and is running entirely out of local RAM, we satisfy the "statelessness" and "locality" principles. Since the Blue Gene control system allows us to scalably boot the same image across nodes, we have a controlled form of the "commonality" principle. Since we can checksum and archive known-good images, we built a rudimentary version control system that tracks the lineage of images, which satisfies the "auditability" principle.

5.2 Applications

We experimented with a diverse set of applications packaged as software appliances to test our hypothesis that the Blue Gene platform can be used for a wide variety of usage cases. We primarily focused on a class of applications that fit into the Web 2.0 paradigm or are requirement thereof. In the following we briefly discuss some of our test applications:

SpecJBB: SPECjbb2005 [17] is a Java benchmark that measures the number of business operations a machine can sustain. The benchmark has a multi-JVM mode to deal with non-scalable JVMs. We used this mode and were able to spread the load across 256 Blue Gene nodes[3] by using a harness that transparently forwards the network and filesystem accesses made by each worker.

We were able to run the benchmark across the 256 nodes that were available to us with a per-node performance of 9565 Business Operations per second (BOPS), yielding a reported score of 2.4 million BOPS. It is important to note that the benchmark rules state a requirement of a single operating system image, so we are not able to submit our performance results at this point. However, our initial results show that Blue Gene/P provides a powerful generic platform to run complex workloads.

Web 2.0: We experimented with Web 2.0 applications that are typically constructed from a LAMP stack (that is Linux, Apache, MySQL and PHP). We package the PHP business logic and Apache webserver in a 20MB appliance. By separating the database from the rest of the application stack, the nodes remain stateless. It is interesting to note that once the cost has been paid to parallelize a workload, the performance of individual nodes becomes irrelevant compared to overall throughput and efficiency. Since web programmers are implicitly forced to parallelize their programs through the use of stateless business and display logic, their workloads make a good fit for an efficient highly parallel machine like Blue Gene. It is also important for the survival of a web company that suddenly becomes popular to be able to quickly scale their capacity, something that is difficult to do with commodity hardware that can require weeks of integration effort to bring online an additional thousand nodes. In contrast, a Blue Gene rack of 1024 nodes is validated during manufacture as a single system.

System S: System S [13] is a stream-processing system that can utilize a diverse set of platforms. Nodes can be dynamically added and removed from a collective and a workload balancer places jobs on the available compute resources. Our infrastructure naturally fits the System S paradigm and we are testing System S on Blue Gene on a number of nodes that it had never run on before.

This example revealed another important aspect of the availability of a consolidated platform: it is now possible to test workloads at scales of tens of thousands of nodes and reveal and eliminate scalability bottlenecks which are not apparent at scales of a few hundred nodes. There is significant scientific value in being able to run at scale, a capability that has usually been reserved for those able to port their code to specialized kernels.

Compute Farm: The time required to compile large software projects may severely hinder development productivity. For some projects groups use thousands of underutilized servers and developer workstations to run distributed compilation, number crunching, and simulation jobs. Capacity at scale significantly reduces the turn-around time and the resources can be efficiently shared between multiple organizations. For example, we are experimenting with running `distcc` on hundreds of nodes to reduce compile times. Developers simply point their make logic to a farm of standby nodes which process the compile requests in parallel.

Storage: Blue Gene/P has no inherent permanent storage, but processors and memory are plentiful. A single rack contains 2 TB of memory. As a first experiment we use some nodes to act as ramdisks; a volume manager then stripes across multiple ramdisks and provides the aggregate storage capacity, for example via NFS. If persistence is a requirement we use external storage attached as closely as possible to the 10G Ethernet link. A trickle-out model with a hierarchical storage system (local RAM, remote RAM, external storage) is a possibility we consider, however, we also consider more sophisticated storage models an important area of future research. We have experimented with using BitTorrent as a peer-to-peer distribution model for loading large software appliances from permanent storage.

6. SUMMARY

In this paper we described the vision and exploration of Project Kittyhawk, an ongoing effort at IBM Research which explores the construction of a next-generation compute platform capable of simultaneously hosting many web-scale workloads. At scales of potentially millions of connected computers, efficient provisioning, powering, cooling, and management are paramount. Some of these problems have been successfully addressed by the super-computer community and we are leveraging the scientific investments into IBM's Blue Gene/P super-computer platform, a system which is specifically designed to scale to hundreds of thousands of nodes as is now demanded by constantly and quickly growing Internet services.

Our early results are promising and show that it is indeed feasible to construct flexible services on top of a system such as Blue Gene. Our design uses the system's high-speed networks to provide internal and external connectivity, combines the significant memory capacity with access to Ethernet-attached storage for low-latency permanent storage, and groups processors into networks of collaborating machines. Our envisaged model is where compute resources are provisioned and used in highly flexible manners. Therefore, in our prototype we attempt to minimize the restrictions to maximize the flexibility for users. To test our hypothesis, we are prototyping a stack consisting of a network-enabled firmware layer to bootstrap nodes, the L4 hypervisor for partitioning and security enforcement, Linux as a

[3]The availablity of Blue Gene/P resources to us is still fluctuating and limited at this early point in the product cycle.

standard operating system, and an efficient software packaging and provisioning system. An important aspect is that while these building blocks allow us to run a large variety of standard workloads, none of these components are required and therefore can be replaced as necessary to accommodate many diverse workloads. This flexibility, efficiency, and unprecedented scale makes Blue Gene a powerhouse for running computation at Internet scale.

7. ACKNOWLEDGEMENTS

We would like to acknowledge and thank the many researchers, over many years, who have contributed to making Blue Gene a reality. The Blue Gene effort incorporates advances in many diverse areas of super-computing including, architecture, networking, packaging, manufacturing, control systems and applications. Our work in exploring a global-scale, general purpose, computational platform has been dramatically accelerated by the existence and access to Blue Gene. We would like to explicitly thank Jose Moreira, Jim Sexton, Robert Wisniewski, Mark Giampapa, and Dilma Da Silva for their help and support.

8. REFERENCES

[1] Appavoo, J., Silva, D. D., Krieger, O., Auslander, M., Ostrowski, M., Rosenburg, B., Waterland, A., Wisniewski, R. W., Xenidis, J., Stumm, M., and Soares, L. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems (TOCS) 25*, 3 (2007), 6.

[2] Carter, J. B., Khandekar, D., and Kamb, L. Distributed shared memory: Where we are and where we should be headed. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (1995).

[3] Coraid. *EtherDrive Storage*. http://coraid.com.

[4] Daly, D., Choi, J. H., Moreira, J. E., and Waterland, A. Base operating system provisioning and bringup for a commercial supercomputer. In *International Parallel and Distributed Processing Symposium (IPDPS)* (2007), IEEE.

[5] Denx Software Engineering. *Das U-Boot – the Universal Boot Loader*. http://www.denx.de/wiki/UBoot.

[6] Dunkels, A. *lwIP – A Lightweight TCP/IP stack*. http://www.sics.se/~adam/lwip/.

[7] Dunkels, A. Full TCP/IP for 8-bit architectures. In *The International Conference on Mobile Systems, Applications, and Services (MobiSys)* (San Francisco, CA, May 2003), USENIX.

[8] Fan, X., Weber, W.-D., and Barroso, L. A. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)* (New York, NY, USA, 2007), ACM Press.

[9] Goldberg, R. P. Survey of virtual machine research. *IEEE Computer Magazine 7*, 6 (1974).

[10] Grbic, A., Brown, S., Caranci, S., Grindley, G., Gusat, M., Lemieux, G., Loveless, K., Manjikian, N., Srbljic, S., Stumm, M., Vranesic, Z., and Zilic, Z. Design and implementation of the NUMAchine multiprocessor. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)* (Los Alamitos, CA, June 15–19 1998), ACM/IEEE.

[11] IBM. Exploiting the Dual Floating Point Units in Blue Gene/L. White Paper 7007511, IBM, http://www-1.ibm.com/support/docview.wss?uid=swg27007511, June 2006.

[12] IEEE. *1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture–Description*. IEEE, New York, NY, USA, 1990.

[13] Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)* (New York, NY, USA, 2006), ACM Press.

[14] Li, K. IVY: A shared virtual memory system for parallel computing. *Proceedings of the 1988 International Conference on Parallel Processing, Vol. II Software* (Aug. 1988).

[15] Liedtke, J. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)* (Copper Mountain Resort, CO, Dec. 1995).

[16] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP '07)* (Brighton, UK, Oct. 2005), ACM.

[17] The Standard Performance Evaluation Corporation (SPEC). *SPECjbb2005 Java Server Benchmark*. http://www.spec.org/jbb2005.

[18] Vise, D., and Malseed, M. *The Google Story: Inside the Hottest Business, Media, and Technology Success of Our Time*. Delta, Aug. 2006.