first published in: HPDC '10, Illinois, USA, June 21-25, 2010. ISBN 978-1-60558-942-8

Providing a Cloud Network Infrastructure on a Supercomputer

Jonathan Appavoo[¶] Volkmar Uhlig[|] Jan Stoess[§]
Amos Waterland[†] Bryan Rosenburg[†] Robert Wisniewski[†]
Dilma Da Silva[†] Eric Van Hensbergen[†] Udo Steinberg[‡]

[¶]Boston University
[|]Aster Data Inc.
[§]Karlsruhe Institute of Technology, Germany
[†]IBM Research
[‡]Technische Universität Dresden, Germany

ABSTRACT

Supercomputers and clouds both strive to make a large number of computing cores available for computation. More recently, similar objectives such as low-power, manageability at scale, and low cost of ownership are driving a more converged hardware and software. Challenges remain, however, of which one is that current cloud infrastructure does not yield the performance sought by many scientific applications. A source of the performance loss comes from virtualization and virtualization of the network in particular. This paper provides an introduction and analysis of a hybrid supercomputer software infrastructure, which allows direct hardware access to the communication hardware for the necessary components while providing the standard elastic cloud infrastructure for other components.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems

General Terms

Management, Design, Measurement, Performance

Keywords

Supercomputing infrastructure as a service, supercomputer network models, user-level networking, high performance cloud computing, high performance computing

1. INTRODUCTION

Huge numbers of processors and massive communication infrastructure are characteristics common to both supercomputers and cloud computing [9] systems. Cost and scale considerations are driving cloud system hardware to be more highly integrated, that is, to more closely resemble today's supercomputer hardware. This trend motivates the effort described in this paper to demonstrate the feasibility of using of supercomputers as cloud infrastructure. Our goal is to support the dynamic usage model associated with cloud computing and at the same time preserve the supercomputer aspects that support high performance applications.

In many ways datacenters are the "computers" behind cloud computing [31, 11]. The typical approach to datacenter infrastructure for cloud computing is to build a collection of general purpose commercial servers connected by standard local-area network (LAN) switching technologies. On top of this an Infrastructure as a Service (IaaS) [36] model is constructed where users are given access to the system's capacity in the form of virtual servers and networks. Platform virtualization software such as Xen is thereby widely regarded as the key enabler for IaaS, as it provides users with a standard software environment *they* know and control, and administrators with a multi-tenancy usage model that achieves high utilization of the datacenter's resources.

Both Cloud computing environments and supercomputers are designed to support multiple independent users who are not the owners of the machine but use some fraction of it for their jobs. While comparable in size, however, the infrastructure for scientific computing is quite different from cloud computing infrastructures. Supercomputers often rely on customized processing elements and integrated interconnects to deliver high performance in computation and communication. Moreover, the supercomputer approach has been to use hardware-enforced coarse-grain partitioning rather than virtualization in order to help user applications achieve more predictable high performance through isolated and dedicated use of resources. Historically, also supercomputing software was highly customized to exploit specific hardware features only present on supercomputer hardware in order to obtain the best possible application performance. More recently, however, the trend has been to adopt standard general-purpose systems software environments in order to ease application development and facilitate portability.

It might be natural to assume that this trend would imply that using a typical IaaS such as Amazon's EC2 for HPC computing would give scientific users the advantages of elasticity while maintaining performance. Unfortunately, although underlying software models have converged, there is evidence that the lack of dedicated access to the hardware and fine-grained sharing of resources as-

.

sociated with virtualization makes performance prohibitively poor. Napper et. al found significant performance degradation when attempting to run Linpack on EC2 [22]. Want et. al studied the effects of virtualization on communication in EC2 and found significant degradation in performance [33]. In particular, and independent of network contention, CPU virtualization can significantly degrade performance of communication sensitive applications.

To summarize, a central feature of cloud computing is elastic resource provisioning of standardized resources. Specifically, users can allocate and de-allocate capacity from a large consolidated pool of third party computing resources as the needs of their applications change during runtime. Elasticity allows them to dynamically scale, both up and down, their usage and costs with their actual needs for each phase of computation. To date, cloud computing has focused on commercial models of computing, based on general-purpose workloads and networking. It has achieved standardization and high levels of elasticity through virtualization, but allows little or no direct hardware access, hinders specializations, and limits predictability and performance. In contrast, scientific computing has exploited specialized infrastructure, including non-standard communications networks, to achieve predictable high performance through direct hardware access and physical dedication but with little or no elasticity.

Elastic resource provisioning, however, and its associated dynamic multi-tenancy usage model, constitute a foundation for scaling out computers to true global dimensions (i.e., across companies, institutions, or even countries), independent of whether the underlying architecture is geared towards cloud or supercomputer infrastructure. Moreover, even many high-performance applications could exploit elasticity, since they can grow and shrink resource usage dynamically based on how their algorithms are proceeding. Elasticity can therefore provide benefits also to high-performance systems, which leads us to two hypotheses that underlie this work:

1. Elasticity can be achieved along with the predictability associated with high performance computing.

2. General-purpose elastic computing can be supported with direct access to specialized networks.

Our work shows how to validate these hypotheses by adding a form of dynamic network virtualization to a supercomputing system. To do so, we propose the construction of an appropriate supercomputer software infrastructure that concurrently supports custom and general-purpose networking protocols in the context of a commodity operating system layer. Specifically, the contributions of this paper are to:

- introduce *communication domains* as a basic mechanism for virtualizing a supercomputer interconnect into dynamically-sized and user-controlled communications groups thereby enabling elasticity without requiring full platform virtualization,

- prototype general-purpose networking, in the form of Ethernet topologies, on top of a supercomputer's interconnects, thus enabling a commodity runtime to seamlessly operate within communication domains,

- allow, concurrently, performance-sensitive applications to directly exploit the communication facilities.

This paper focuses on describing how our prototype system implements both general purpose communication, in the form of an Ethernet implementation, and direct hardware-based communication on top of the elasticity provided by communication domains. The paper is structured as follows: Section 2 discusses how we designed communication domains as a mechanism for introducing elasticity on a supercomputing platform. Section 3 shows how we map a typical Ethernet-based cloud cluster network model on to the Blue Gene/P supercomputer. Section 4 describes how we provide concurrent access to the interconnect hardware features in the context of a specific application scenario. Finally, Section 5 presents our evaluation, followed by related work in Section 6 and a summary in Section 7.

2. COMMUNICATION DOMAINS

To allow for *both* elastic and predictable IaaS provisioning on a single system, we propose a new mechanism, called a communication domain, that provides an abstraction to flexibly express communication permissions without restricting direct hardware access. By definition, a communication domain is a set of nodes that are permitted to communicate with one another. A node denotes a compute element of a supercomputer combining CPU, memory, and a physical location in the interconnect networks. A node may belong to more than one domain. Each domain to which a node belongs is manifested on the node as an interconnect interface, whose implementation is unspecified. However, we assume that the consolidated nature of the system will result in interconnects that support communication modes akin to integrated system buses, thus enabling a wide range of protocol and communication models such as Ethernet to be explored.

Our implementation of communication domains enables users to grow or shrink, at any arbitrary time, the number of nodes in a domain through a standard resource allocation interface. Critical to the concept of communication domains is the underlying feature of a supercomputer: an interconnect that provides a flat uniform physical communication name space. In this way, every node is assigned a unique physical identifier (e.g., its position in the physical topology of the interconnect), and can potentially reach any other node by sending a message to the node. We then layer the logical construct of a communication domain atop, to dynamically and efficiently construct groups of nodes that have logically restricted rights to communicate.

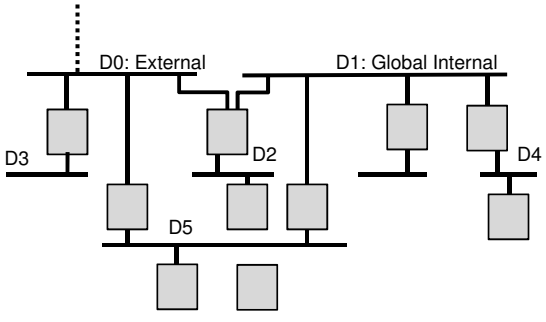


Figure 1: Example of a topology of communication domains.

Note that very similar approaches are currently being proposed for commercial IaaS infrastructures [17, 24], albeit as a retro-fit onto commodity interconnects. These approaches attempt to introduce a new ubiquitous identifier for datacenter networks that is below the standard layer 2 MAC address and encodes a physical location in the datacenter on top of which a restricted logical MAC

address mapping is constructed. Given the commodity nature of these systems, software is restricted from anything other than MAC address access and the extra layer is implemented as extensions to the commodity switching hardware. While such an approach allows for more dynamic organization of the Ethernet topologies, it does not permit higher-performance network access, and still restricts software to continue to use Ethernet protocols for communication.

Communication domains stay in contrast to full platform virtualization, which is the typical mechanism to achieve CPU and network isolation in cloud computing: virtualization requires a hypervisor to run on each node, and, by design, separates users from the hardware. We do not preclude full virtualization from our architecture, however; rather, we propose to absolve it from its duty as first-class isolation mechanism, in favor of the more bare-bones communication domain concept, which also has less (potential) implications on end-to-end performance. In many ways communication domains are more closely related to Virtual Private Network (VPN) [34] support in commodity switching hardware. Indeed, some of today's cloud IaaS providers have recently begun to support user VPN management, in order to restrict communication logically. However, unlike VPN technology, communication domains are not bound to the use of Ethernet or IP protocols and can be mapped more easily to the distributed nature of supercomputer interconnects and are designed for fast scalable user control.

The rationale behind communication domains is twofold: achieving dynamic resource management without sacrificing performance and predictability. We posit the degree of dynamic flexibility of communication domains enables us to address both the elasticity and predictability goals in resource management of our system.

On the one hand, the communication domain allows infrastructure users to dynamically grow and shrink their resource consumption. A user can create domains and dynamically allocate nodes into the domains to construct arbitrary permission topologies. For example, a user might create an isolated communication domain to restrict communication to only be permitted among those nodes. Another user might create two domains, placing nodes in either or both. The nodes in both domains can act as gateways between the two. In a similar spirit, an IaaS offering can provide well-known system domains that are publicly accessible, allowing multiple users to communicate with each other by placing nodes within them.

On the other hand, communication domains also permit raw access to physical resources for at least some applications running on the infrastructure. The communication domain abstraction does not make any assumptions about the software running on nodes; thus it naturally permits users to gain access to CPU (and memory) resources, if this is desired. No restriction is placed on the software: some users may choose to run virtualization software, thereby providing a flexible and potentially low-cost standard IaaS environment that is dynamically composed from some portion of the supercomputer resources. Other users may choose to run their performance-sensitive applications on a dedicated set of nodes, in order to sustain the high and guaranteed delivery of processing power the applications typically expect. However, more dynamic application models for high-performance scenarios can be explored. For instance, a user running a tree-based search algorithm can adapt the number of nodes used dynamically, based on how the tree search is proceeding.

Similarly, the nodes within a domain are free to choose to communicate using any protocol they like: this could range from MPI instances which utilize the hardware interface directly to software implementations of a protocol such as Ethernet. Moreover, while the communication domain does not imply physical placement with respect to network interconnects it also does not restrict users from trading nodes to create pools that do have good placement. By definition, the communication domain model does not imply physical topology, and nodes that are dynamically allocated by a user can have any physical location on the interconnects of the system. However, the user still has raw access to the nodes in the sense that she can run arbitrary software, including software that can identify the nodes' physical location and determine network proximity or lack thereof. That way, users or administrators can construct domains and allocate nodes in a way that allows utilization of topology information to sustain top network performance. One could also imagine users that precisely construct services that provide nodes with good locality for other users that require it. While the locality guarantees may not be equivalent to having an electrically isolated block of nodes, it makes it possible to imagine a very large installation that has many diverse users running applications with many different requirements.

3. MAPPING A CLOUD NETWORK ONTO A SUPERCOMPUTER

In this section, we describe how we prototyped an elastic supercomputing model by adding a cloud-like networking layer to a supercomputer. A typical commodity cloud cluster is built out of hosts that run a commodity OS and are interconnected by a physical network. The operating systems most often communicate via Ethernet, both among themselves and to the networks exterior to the cluster. Clusters also tend to have some form of storage infrastructure, which can range from local disk drives to advanced Storage Area Network devices and combinations of both. In order to support such a cluster model we map the common system abstractions onto the Blue Gene/P supercomputer.

3.1 Blue Gene/P Overview

The basic building block of Blue Gene/P (BG/P) is a node composed of a quad-core processor derived from the embedded PowerPC 440 core, five networks, and a DDR2 memory controller integrated into a system-on-a-chip. Each node further contains 2 or 4 GB of RAM soldered onto the node for reliability reasons. The nodes are grouped 32 to a node card, and the node cards are grouped 16 to a mid-plane. There are 2 mid-planes in a rack, providing a total of 1024 nodes and, in its larger configuration, a total of 4 TB of RAM. Multiple racks can be joined together to construct an installation. The system supports up to 256 racks totaling over 1 million cores and 1 Petabyte of RAM. Blue Gene/P features the following three key communication networks:

Torus: The torus network [6] is the most important data transport network with respect to bisectional bandwidth, latency, and software overhead. Each compute node is part of the three-dimensional torus network spanning the whole installation and is identified by its X,Y,Z coordinates. Each node has input and output links for each of its six neighbors, for a total of 12 links, each with a bandwidth of 3.4 Gbit/s (425 MB/s), for a total node bandwidth of 40.8Gbit/s (5.1 GB/s). Worst-case end-to-end latency in a 64 K server system is below 5μs. Nodes in the network act as forwarding routers without software intervention. The torus provides two transmission interfaces, a regular buffer-based one and one based on remote direct memory access (RDMA).

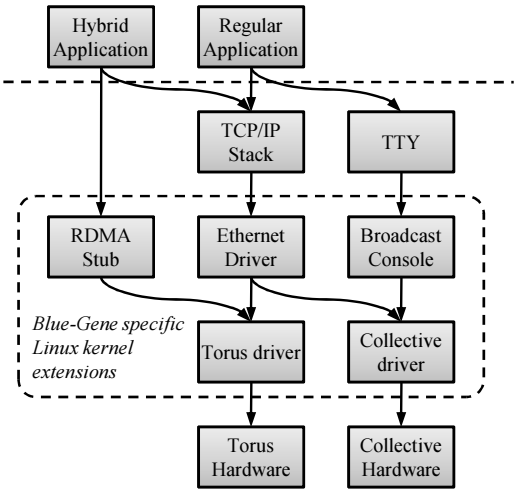


Figure 2: Blue Gene communication stack in Linux. We separate the low-level links and link-encapsulation from the higher-level protocols such as Ethernet and TTYs. Additional protocols, such as SCSI, ATA or USB can be layered as well.

Collective: The collective network is an over-connected binary tree that spans the installation. The collective is a broadcast medium with node-specific filtering and a complex routing scheme that allows for sub-partitioning the network to increase the total bandwidth. The collective link bandwidth is 6.8 Gbit/s (850 MB/s), and packet delivery is in-order and guaranteed in hardware.

External Ethernet: Additionally, each node card can have up to two I/O nodes, which feature the same basic unit as the compute nodes but replace the torus device with an on-die 10 Gbit/s XFP Ethernet controller. When fully populated each rack has an external I/O bandwidth of up to 640 Gbit/s.

3.2 Ethernet on Blue Gene/P

To support a standard software environment in a cloud or utility computing model, our prototype currently provides a BG/P-specific version of the Linux Kernel. The internal network model we support is a hybrid of Ethernet for legacy applications and a kernel bypass with direct access to the specialized hardware from application level. (We will explain details on the direct hardware in the following Section.) To facilitate Ethernet communication, the prototype features a set of Linux device drivers that map an Ethernet-based version of our communication domain model (Section 2) onto the dedicated BG/P interconnects. When using the drivers, each communication domain appears as a separate Ethernet network. Our initial choice was to abstract at the IP layer and map protocol specifics to the hardware features, but we quickly learned that the world speaks Ethernet and not IP. We further support a kernel bypass with direct access to the specialized hardware from application level, in order to exploit tight coupling (Section 4).

In our architecture we separated the low-level device driver and packetization engine from the device abstractions such as Ethernet (see Figure 3.2). High-level drivers can register wire protocols on the torus and collective. Thus we can multiplex arbitrary protocols over the same underlying network and driver infrastructure. A unique Ethernet interface is instantiated for each configured communication domain.

In our standard Linux environment we primarily use Ethernet and a multicast console exported as a TTY. Point-to-point Ethernet packets are delivered over the torus and multicast packets over the collective network. We chose a MAC address scheme such that MAC addresses directly convert into torus coordinates and thus the torus becomes a very large distributed switch. I/O nodes act as transparent Ethernet bridges that forward packets between the external world and authorized compute nodes. I/O nodes are not members of the torus and instead use the collective to reach the compute nodes.

BG/P's networks are reliable, a feature we exploit. For all internal traffic we eliminate network checksumming. When a packet leaves the system the bridge on the I/O node inserts the checksum using the hardware offload engine of the 10 GBit/s adapter. For the console we rely on guaranteed in-order delivery which significantly simplifies the implementation of group communication, in particular in the early stages of a boot with thousands of nodes.

4. EXPLOITING TIGHT COUPLING

Our Ethernet virtualization layer enables the standard networking environment that is typically delivered in today's IaaS offerings. However, to provide communication services of high predictability and performance, as required in typical HPC applications, we also allow the distributed switching capabilities of BG/P's tightly integrated interconnect to be used in a more direct way.

Our architecture strives to provide a development path from standard to specialized technology: on the one hand, it enables extending existing distributed systems software to produce an environment compatible enough to host existing large-scale distributed applications. On the other hand, it permits customizations that exploit the hardware features of the tightly-coupled interconnect. To prototype such a use case, we customized a general-purpose distributed memory caching system called memcached (memory cache daemon) [15] to utilize the RDMA features of BG/P hardware; we will describe memcached and our specializations in the following.

4.1 Memcached

Memcached is a distributed memory object caching system often used to speed up web applications by caching database query results in memory, with the goal of alleviating database load and contention [15]. Memcached can be viewed as a generic cache service, in which a set of servers located on TCP/IP-accessible hosts cooperate to implement a unified in-memory cache. The cache is accessed by application hosts through a client library that communicates with the servers via TCP/IP. Several client libraries exist. In our prototype we use the common libmemcached client library, a C and C++ client library designed to be lightweight and to provide full access to the server-side methods [2]. The primary operations of libmemcached are:

memcached_set (set): stores a key-value pair into the cache; keys can be up to 250 B, data up to 1 MB.

memcached_get (get): queries the cache for a particular key; if found, returns the corresponding value.

memcached_delete (del): removes a key-value pair from the cache.

Memcached does not provide authentication or security, or redundancy or increased data availability. Rather, memcached provides a simple, single key-space that can span multiple servers to cache a data object in the memory of one of the servers. Its primary goal is to improve performance by alleviating load and contention on the back-end data repository, typically a data base. A wide range

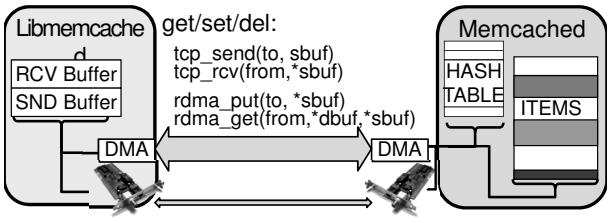


Figure 3: Memcached primitives implemented via TCP/IP and BG/P RDMA

of existing Internet services use memcached, including large-scale web-based services such as Slashdot, LiveJournal, and Facebook.

From our perspective, memcached serves as an accepted simple building block for distributed service construction. Our goal is to preserve its interface, while modifying its internals to utilize the features of BG/P and improve its baseline performance. Specifically, we modified memcached to use torus RDMA to move data directly between nodes, eliminating protocol stack overheads, while preserving the interface and semantics of memcached. We will describe our approach in the following; it has led to a 2.2x–9.7x increase in the get performance of memcached over the default TCP/IP based implementation; for 64 KByte buffers, it achieves 80% of the hardware's theoretical maximum throughput.

4.2 Memcached and Blue Gene/P RDMA

Figure 3 illustrates the two components of memcached; the client library on the application host and the memcached server on the server host. In the default implementation the two utilize a protocol transmitted over TCP/IP to jointly implement the **set**, **get** and **del** operations[1].

Our memcached specialization consists of two parts: our first performance improvement was to separate the control and data path of memcached and to use RDMA to transmit the payloads directly between client and server, effectively eliminating the overhead of TCP/IP. The second optimization was to split the meta-data structure of the hash table from the payloads, and to allow clients to maintain a local copy of the server's hash table, if the consistency model permits it. Using the local copy for look-up, clients can directly access the server's memory avoiding the overhead and extra latency of a control message. With our performance optimizations the mechanics of memcached **get**, **set**, and **del** are as follows:

Get: The get operation retrieves an item from a set of memcached backend servers. The client library first determines the responsible server based on the key and then initiates a network request to fetch the item.

We replaced the standard TCP/IP implementation of the data path with an implementation using RDMA. Each server allocates a large RDMA-addressable memory chunk and exports it for use over the torus network. All allocations for control structures and key-value pairs are served from that allocation so that a client can directly access server memory. The server also publishes the hash table containing the key-to-item mappings. The server stores its items in a collision chain with the key and value stored consecutively in memory.

To retrieve an item over RDMA, the client initiates two requests, one for the hash table and a second request for the data. Further requests may be necessary if the server has a long collision chain. Depending on the data consistency model, the client can reduce the number of RDMA accesses by caching the hash table exported by the server. When a key is referenced but it cannot be found in the locally cached table, the client re-fetches the table and retries the lookup. If the key is still not present, the library reports a miss to the higher layers. When a reference is found in the local cache, the client fetches the data item without updating the hash table, thereby avoiding the extra RDMA operation. Since the key and versioning information can be stored inside the data block, the client can determine if the copied item is stale or still valid. If the item is invalid, the client re-fetches the hash table and retries the process.

Set: The set operation stores an item in a memcached server. Memcached supports three variants of this operation to implement cross-client synchronization. The **add** operation adds the item if it does not exist in memcached; the **replace** operation modifies the key-value if it exists; and the **set** operation updates the key independent of whether it existed beforehand or not.

All operations require serialization on the server side in order to perform the existence check (if necessary), allocate memory for the key-value item, and register the item in the hash table. We extended memcached with a two-phase protocol, one carrying the control traffic and the second carrying the data traffic. For the control traffic we use standard TCP/IP messages; such control messages contain an RDMA reference to allow the *server* to initiate an RDMA request if item data needs update (i.e., if it is not an add operation). In this case, the server issues an RDMA fetch from the client, after allocating the backing store for the item. The two-phase approach has a higher overhead than the pure RDMA-based one of **get**, due to the additional latency; but it reduces the copy overhead on the client and server for sending and receiving the data through sockets. In our measurements for payload sizes of 1024 bytes the two-phase approach has about the same cost as transmitting the payload with the control message.

Del: The delete operation removes a previously stored item from the hash table, unlinks it from the collision chain and frees the used memory. Since we are replicating the hash table in the clients it is not sufficient to just unlink the item from the hash table since stale cached copies may still reference the data item. The **del** operation therefore first marks the item invalid in the data item, then unlinks it from the hash table and finally frees the memory.

Coherency and RDMA

The standard implementation of memcached uses a strict coherency model with full serialization in the server using TCP/IP and a single-threaded message loop. Maintaining the strict consistency eliminates most optimization potential for RDMA and limits it to the data path, reducing the overhead of multiple memory copies. We therefore relaxed the original consistency model and leave it up to the users of our memcached implementation to develop their own consistency models atop, which renders them free to apply their domain-specific knowledge about consistency requirements of particular workloads.

In our case, we used memcached as a front-end for content-addressable storage, where each key is a secure hash of its corresponding value, thus each key will always have exactly one associated data item. This domain-specific knowledge allowed using an all-or-nothing approach for the data item: Data is either correct or incorrect, since the fixed binding of key to value eliminates the situation where a client may read a partially modified item. Similarly,

[1]There also exists a UDP-based prototype for the memcached back-end. However, the libmemcached version we use proved to be not fully functional for UDP, thus we resorted to the original TCP/IP implementation.

we can easily resolve conflicts due to staleness of local hash-table copies: A client may have a stale hash table and fetch an item from memory which was reused. Since key and value are always paired we do not need to worry about stale accesses since the client will be able to detect a conflict.

If a stricter consistency model is required, we propose to use version numbers and a double-check algorithm that re-fetches the valid bit after having fetched the complete value data. BG/P's torus RDMA engine can handle multiple, ordered, RDMA fetch requests to different target segments, thus the re-fetch can be piggybacked onto the original data fetch, avoiding an additional round-trip message overhead.

5. PERFORMANCE RESULTS

In this section, we present experimental results we obtained from our prototype. We will first present a performance validation of the Ethernet network layer running atop the BG/P interconnects. We then present a performance validation of our BG/P-customized memcached implementation.

Note that the following results are preliminary: we did all our experiments on a contended BG/P system, and at the time of benchmarking, the largest available allocation size was 512 nodes; also, although we ran all experiments several times, we did not collect deviations and only list average results. We are in the process of trying to bring our prototype up at a new installation and expect to present an extensive performance study in a subsequent publication.

5.1 Network Performance

BG/P is architected as a system balanced with respect to computation and communication. A key design criterion for the system was to provide network bandwidth sufficient to let all processors transport data either to peers or to external storage arrays.

We ran two experiments. The first experiment measures the network performance from an external host to one compute node. The packets travel through one I/O node and get forwarded over the collective network to the compute node using Linux' bridging module. The second experiment measures the performance between two compute nodes using the torus network. The benchmark application is netperf [3] which consists of client and server programs, each single-threaded.

We measured the network throughput for both setups and varied the packet sizes and the network protocols (TCP and UDP). The results are listed in Figure 4. We also provide baseline performance running both client and server on the same node (but different cores) communicating through the loopback interface. In this scenario Linux just copies the data from the netperf client to the kernel, processes the packet in the IP stack, and copies the received packet back to the netperf server. The results show that the IP stack imposes a significant overhead.

Several factors influence the realized torus TCP and UDP performance, which falls significantly short of the maximum 3.4 Gbit/s a single torus link can provide. First, note that UDP over the loopback interface with 8000-byte packets achieves approximately 2.2 Gbit/s. This serves as an upper bound on what one could hope to achieve, as both the IP protocol processing is minimal and all data is being transfered via local memory copies. The rest of this discussion will focus on the 8000-byte packet results.

As indicated in the table, the UDP result between two torus nodes achieves about 1.1 Gbit/s, or approximately half the loopback throughput. Although at this time we do not have a precise breakdown of the costs, it is worth highlighting some of the details. Keep in mind that although the upper layers of Linux interact with our driver as an Ethernet device, the underlying driver must work with the hardware parameters of the torus. This means that regardless of the Ethernet Maximum Transmission Unit (MTU) settings, our implementation must send and recieve data with torus hardware packets that are at most 240 bytes. In our experimental environment we use an MTU size of 9000 bytes, which is larger than any of the packets we use in our netperf experiments. While this minimizes the number of Ethernet packets and the Linux overhead associated with Ethernet packet processing, our torus Ethernet implementation must still process the individual Ethernet frames in 240-byte fragments. This processing has overhead that includes interrupt handling, frame reassembly, and copying that are not present in the loopback results.

TCP processing adds additional overhead, as the TCP window size limits the number of bytes that a sender will transmit before waiting for an acknowledgement. This increases overhead because it throttles the sender and potentially increases the number of packets that must be both sent and received. The anomaly of TCP being faster than UDP for small packets can be attributed to the TCP packet buffering mechanism in the kernel.

One can certainly imagine optimizations that exploit semantic knowledge to map TCP/IP communication more effectively to the capabilities and parameters of the torus interconnect. For example, given the reliable nature of the torus one could avoid, in certain scenarios, the need for acknowledgements altogether. We chose not to focus on such optimizations. Our goal is a straightforward and transparent Ethernet-based TCP/IP implementation that makes it easy for users to initially use the system. As performance becomes a greater focus, we expect users to exploit the hardware capabilities directly with application-specific protocols and communication facilities and only use TCP/IP and Ethernet for non-critical functions. Furthermore, our model is not bound to a single implementation of Ethernet and TCP/IP. Alternatives can be easily developed and explored.

In addition to the previous scenarios, we measured the latencies for transmitting the data directly using a user-level RDMA layer and bypassing the kernel. The results are shown in the rightmost column of Figure 4. The latency results for the loopback tests are dominated by the cross-processor signalling overhead; the Linux scheduler places the netperf client and server programs on different cores. With larger packet sizes, the zero-copy loopback path in the Linux kernel incurs a lower overhead than the real network devices, resulting in lower latency.

Overall, the results demonstrate that our Ethernet implementation provides sufficient end-to-end bandwidth and latencies for general-purpose use, with external bandwidth of up to 940 Mbit/s and internal bandwidth of up to 436 Mbit/s for TCP-based communication. The results also show, however, that protocol-processing overhead can have significant impact on throughput and latency from a user-level perspective, and that direct access to RDMA features is necessary for delivering close-to-raw performance at user level.

5.2 Memcached Performance

We evaluated the performance of memcached running on BG/P using the original TCP/IP-based protocol and also our enhanced version using torus RDMA. We obtained the results using a modified version of the *memslap* micro-benchmark, which is part of the original libmemcached client library. The benchmark performs a sequence of memcached get and set operations to read or write a predefined set of fixed-size items. Our modifications are limited to

Link	Size in Bytes	Throughput in Mbit/s TCP	UDP	Latency in μs
External	200	629.35	198.57	53.5
	1000	923.41	812.19	55.8
	8000	940.56	1832.11	129.0
Torus (Eth)	200	210.53	113.67	33.4
	1000	373.25	500.67	40.7
	8000	436.25	1099.02	162.3
Torus (RDMA)	200	n/a	n/a	1.7
	1000	n/a	n/a	5.3
Loop	200	422.89	71.18	51.7
	1000	806.72	323.92	53.4
	8000	1440.28	2186.23	81.4

Figure 4: Network throughput and latency for external, internal, and node-local communication with varying packet sizes over Ethernet. Additionally, latency for RDMA over torus without kernel involvement.

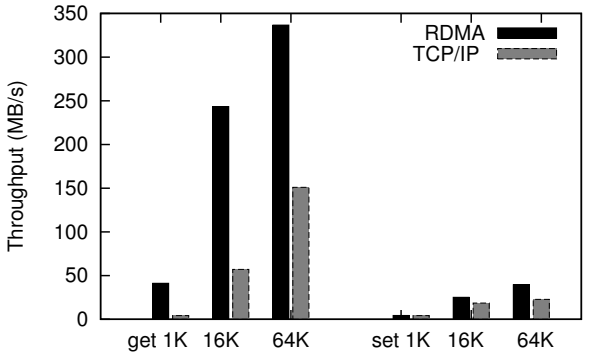


Figure 5: Throughput of memcached get/set primitives implemented via TCP/IP and BG/P RDMA, for different item sizes

providing support for more detailed measurements of operations. Both item keys and values are randomly generated.

Figure 5 compares **get** and **set** throughput of both versions for three different payload sizes, 1 KByte, 16 KByte and 64 KByte, respectively. The results show that the read operation clearly outperforms the write operation, independent of the protocol. The overhead can be attributed to the hash table manipulations. The RDMA performance is significantly higher than with TCP/IP. For 1 KB payloads **get** throughput using RDMA outperforms TCP by an order of magnitude. For large **get** operations, RDMA reaches throughput close to the limits of the interconnect (340MB/s out of a possible 425MB/s), a factor of 2.2 improvement compared with TCP/IP. The **set** operation using RDMA consists of a TCP control packet followed by a remote fetch using RDMA. As expected, the throughput increases with larger payload sizes. We found that RDMA almost always pays off, except for the write case with very small payloads, where the cost of the extra DMA operation outweighs the additional copy overhead through TCP/IP.

5.3 Memcached Scalability

In the next experiment we evaluated the scalability of memcached by varying the number of clients and servers. We ran *memslap* again, this time with varying numbers of clients and servers. We used a unique key to prefix all items of each individual node. We then averaged the bandwidth for all clients. We determined the combination for all configurations of power-of-two for clients and servers up to 512 nodes (at the time of benchmarking the largest available allocation size was 512 nodes, since the machine is in active use by many projects).[2]

Figure 6a and 6b show the throughput for 1K and 64K items in a 3D graph. We expected two trends: First, with increasing number of clients per server the throughput per client should decrease. Second, since the location of clients and servers was chosen at random, the increased load on the links for forwarding traffic should decrease the overall bandwidth. Both trends are clearly visible in the figure.

In Figure 6c we present the same data but keep the ratio of clients to server constant and vary the total number of nodes. We normalize the throughput against the case of one server and one client. With perfect scalability the throughput of each data set would remain constant, forming a horizontal line. Obviously this is not the case and the throughput declines with increasing number of nodes in the system.

As expected, node locality plays an important role in overall performance. The conclusion we draw is that the node allocation system needs to provide an explicit placement mechanism that enables clients to leverage locality. Furthermore, it is important that nodes observe communication patterns and potentially relocate the workload closer to peers. It implies that the system needs to provide secure mechanisms for monitoring and to introduce resource management policies at the installation level, i.e., the datacenter.

6. RELATED WORK

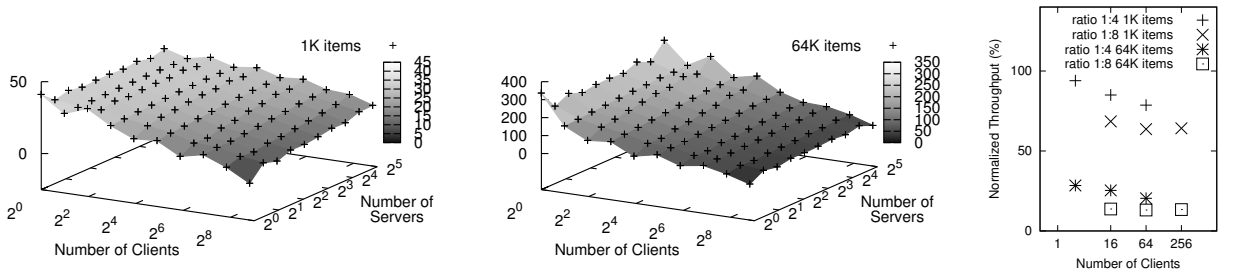
Amazon's EC2 and Microsoft's Azure are examples of cloud computing offerings available today. Although little is published about the platforms, we do see a trend in the products targeting cloud computing, such as Rackable's MicroSlice^{TM}products [4], which, with a total of 264 servers per rack, provides a density that is 3 to 6 times the density of a normal rack. Such systems still rely on standard external Ethernet switch infrastructure for connectivity but we predict that the trends in commercial system integration will follow a trajectory similar to that of supercomputers.

The Cray XT5 QuadCore [1] supercomputer, which as of November 2008 formed the fastest homogeneous and second fastest overall supercomputer installation, shares many properties with Blue Gene/P, including the 3D torus. Given our understanding of this platform we believe our system model would be equally applicable to it.

Hybrid interconnects have been used in both commercial and research general-purpose multi-processor systems composed of memory and a small number of processors that are interconnected by a homogeneous consolidated high performance network [32, 28]. Within this class of systems we are not aware of any that did not impose shared memory. As such all of the systems required operating system software that managed the machine as a single system image [7, 20, 23]. Creating such functional systems able to run standard workloads can take a decade or more [20]. An interesting alternative to constructing a complete OS for shared memory multiprocessors is the construction of a scalable virtual machine monitor as was explored in the Disco project [13].

Similarly, loosely coupled software systems such as Mosix [10], Plan9 [25], and Amoeba [21] use standard networks like Ethernet

[2]Note that for the reported configuration with 512 clients we were only able to use 448 nodes since the remaining 64 nodes were used as memcached servers or to host management software such as a synchronization server. Since the values are averaged the data is still representative as a trend.



(a) throughput for 1K items (b) throughput for 64K items (c) Normalized throughput

Figure 6: Throughput in MB/s of memcached get primitives (RDMA version), for 1K and 64K item size and different numbers of client and servers. The rightmost figure shows how throughput (normalized in this figure) scales with increasing number of clients, if the client-server ratio remains constant.

to couple nodes at the cost of latency and performance. These systems place significant focus on caching, fault handling, distributed resource management, common naming, and sharing of data. Many of these problems are eliminated due to the tight coupling and low-latency communication links in a supercomputer. To maintain compatibility we take a selective approach; we treat the machines as a cluster of independent instances, yet exploit the tight coupling and homogeneity where it is beneficial to the OS and application layers. We do not create the illusion of a single-system image but provide a common management infrastructure to permit access to resources. Systems with interesting characteristics, such as Plan9 [18] can then be layered on top.

In the area of grid computing, efforts such as VNET/Virtuoso [30], IPOP [16], and Violin [27] have investigated network overlays and virtualization as a means to provide a standardized, scalable, and easily accessible interface to communication resources, which can help to establish virtual distributed environments in a shared infrastructure. While similar in spirit and goals, the grid computing approaches target decentralized and heterogeneous grid architectures, which are loosely connected through wide-area networks. Our approach, in contrast, is more focused on the high-performance, tightly-coupled and comparatively homogeneous interconnect technology found in supercomputers. In addition, our techniques target hardware-level mechanisms that do not fix communications protocols or require specific systems software such as TCP/IP or virtual machine monitors.

From a software infrastructure perspective, the original IBM Blue Gene distribution utilizes Linux in a restricted fashion on the I/O nodes of the system. Each I/O node runs a stripped down, in-memory Linux distribution and Blue Gene-specific Linux software such as tools and libraries. In concert with the external IBM control system, this software provides an environment for users to develop and deploy applications on the system. The I/O node software acts as a bridge between the IBM compute node run time (CNK) and the external networks and storage systems. The ZeptoOS [5] project extends this model, enhancing the I/O node software stack [19] to improve performance and permit application customization and optimization of I/O operations. In addition, ZeptoOS provides a robust Linux-based environment for the compute nodes. In particular, care is taken to mitigate OS interference [12] and overheads associated with demand paging [35] so that high performance applications can be deployed on top of Linux on the compute nodes. Our approach, unlike both the original IBM infrastructure model and ZeptoOS, focuses on low-level primitives for the development and deployment of diverse systems software, both open and closed, and general-purpose and high-performance. We introduce a lower-level dynamic and elastic allocation system that allows users to load nodes with arbitrary software. In addition, we provide a prototyped Linux kernel that exploits the direct access to implement both Ethernet and custom communication protocols on the proprietary interconnects. Our goal, however, is not the development of a single Linux kernel or distribution but rather an environment for the development of both open and closed software stacks.

Finally, distributed hash tables such as memcached are widely used in peer-to-peer and Internet-scale environments. A major part of the work is node discovery, naming, and routing due to the dynamic nature of the peers. Pastry [26] and Chord [29] use routing mechanisms that provide a bounded number of hops. Amazon's Dynamo key-value store [14] provides a highly available distributed data store through replication with late reconciliation.

7. CONCLUSION

Supercomputers present an interesting point in the design space with respect to dense integration, low per-node power consumption, datacenter scale, high bisectional network bandwidth, low network latencies, and network capabilities typically found in high-end network infrastructure. However, an esoteric and highly optimized software stack often restricts their use to the domain of high-performance computing.

By providing a commodity system software layer we can run standard workloads at significant scale on Blue Gene/P, a system that scales to hundreds of thousands of nodes. Sharing, however, requires secure partitioning, allocation, and the ability to freely customize the software stack based on each individual's needs. We showed that, with communication domains as basic foundation and a cloud-like network layer atop, such can be achieved also on a supercomputer. Furthermore, by exposing specialized features of the supercomputer such as Blue Gene's networks, hybrid environments can utilize the advanced features of the hardware while still leveraging the existing commodity software stack. We showed this with an optimized version of memcached utilizing the RDMA capabilities of Blue Gene.

In addition to demonstrating supercomputers useful for deploying cloud applications, the infrastructure described in this paper can also be used to evaluate and analyze cloud infrastructure, management, and applications at scales that are incredibly costly on commercial platforms. A single BG/P rack gives us a thousand-node cloud, and our communication domain mechanisms allows us to

specify interesting overlay topologies and properties. Conversely, we are also interested in examining ways that this infrastructure can be used to backfill traditional high-performance computing workloads with cloud cycles during idle periods, extracting additional utilization from existing supercomputer deployments. Backfill provisioning is already an option on several HPC batch-scheduling systems. We would simply apply these mechanisms to provide supplemental cloud allocations that could be used opportunistically.

Tight integration of computation, storage, networking, powering and cooling will be one of the key differentiators for cloud computing systems, a trend that can already be observed today. Different forces but with similar outcomes have led the HPC community to build highly integrated supercomputers. We expect that over the next decade we will see cloud systems that very much resemble the level of integration we have with Blue Gene. Hence, when designing those commercial systems and software stacks we can learn a lot from supercomputers that exist today and experiment with features and scale.

The infrastructure described in this paper is part of the Kittyhawk project [8], which is now open source and available from http://kittyhawk.bu.edu.

8. REFERENCES

[1] *Cray XT5*. `http://www.cray.com/Assets/PDF/products/xt/CrayXT5Brochure.pdf`.

[2] *libmemcached*. `http://tangent.org/552/libmemcached.html`.

[3] *Netperf*. `http://www.netperf.org/netperf/`.

[4] *Rackable MicroSlice™ Architecture and Products*. `http://www.rackable.com/products/microslice.aspx?nid=servers_5`.

[5] *ZeptoOS – The Small Linux for Big Computers*. `http://www.mcs.anl.gov/research/projects/zeptoos/`.

[6] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2/3):265–276, 2005.

[7] D. P. Agrawal and W. E. Alexander. B-HIVE: A heterogeneous, interconnected, versatile and expandable multicomputer system. *ACM Computer Architecture News*, 12(2):7–13, June 1984.

[8] J. Appavoo, V. Uhlig, and A. Waterland. Project Kittyhawk: building a global-scale computer: Blue Gene/P as a generic computing platform. 42(1):77–84, Jan 2008.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[10] A. Barak and R. Wheeler. MOSIX: An integrated multiprocessor UNIX. In *Proc. of the Winter 1989 USENIX Conference*, San Diego, CA., Jan.-Feb. 1989.

[11] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Synthesis Lectures on Computer Architecture*. Morgan & Claypool, 2009.

[12] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[13] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. of the 16th Symposium on Operating System Principles*, Saint Malo, France, Oct. 1997.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, Oct. 2007. ACM.

[15] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[16] A. Ganguly, A. Agrawal, P. O. Boykin, and R. J. Figueiredo. IP over P2P: Enabling self-configuring virtual IP networks for grid computing. In *IPDPS'06: Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*, Rhodes Island, Greece, Apr. 2006. IEEE Computer Society. U. Florida, USA.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev.*, 39(4):51–62, 2009.

[18] E. V. Hensbergen and R. Minnich. System support for many task computing. In *Proc. of the Workshop on Many-Task Computing on Grids and Supercomputers, 2008 (MTAGS 2008)*. IEEE, Nov. 2008.

[19] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. Zoid: I/o-forwarding infrastructure for petascale architectures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162, New York, NY, USA, 2008. ACM.

[20] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proc. of the First European Systems Conference*, Leuven, Belgium, Apr. 2006.

[21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[22] J. Napper and P. Bientinesi. Can cloud computing reach the top500? In *UCHPC-MAW '09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 17–20, New York, NY, USA, 2009. ACM.

[23] S. Neuner. Scaling Linux to new heights: the SGI Altix 3000 system. *Linux Journal*, 106, Feb. 2003.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 39–50, New York, NY, USA, 2009. ACM.

[25] D. Presotto, R. Pike, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9, A distributed system. In *Proc. of the Spring EurOpen'91 Conference*, Tromso, May 1991.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized

object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, 2001.

[27] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38:63–69, 2005.

[28] J. P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *ACM Supercomputing 93*, 1993.

[29] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM 2001 Conference*, Aug.

[30] A. Sundararaj, A. Gupta, and P. Dinda. Dynamic topology adaptation of virtual networks of virtual machines. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, page 8. ACM, 2004.

[31] W. Vogels. *A Head in the Cloud: The Power of Infrastructure as a Service*. `http://www.youtube.com/watch?v=9AS8zzUaO3Y`, 2008.

[32] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, Apr. 1995.

[33] G. Wang and E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM'10: Proceedings of The IEEE Conference on Computer Communications*. IEEE, 2010.

[34] Wikipedia. Virtual private network — wikipedia, the free encyclopedia, 2010. [Online; accessed 12-March-2010].

[35] K. Yoshii, K. Iskra, H. Naik, P. Beckmanm, and P. C. Broekema. Characterizing the performance of big memory on blue gene linux. *Parallel Processing Workshops, International Conference on*, 0:65–72, 2009.

[36] L. Yousef, M. Butrico, and D. DaSilva. Towards a unified ontology of cloud computing. In *GCE08: Proceedings of The IEEE Conference on Computer Communications*. IEEE, 2008.