RC25241 (W1111-212) November 28, 2011
Computer Science

IBM Research Report

HARE: Final Report

Eric Van Hensbergen
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758
USA

Ron Minnich
Sandia National Laboratory

Jim Mckie
Bell Labs.

Charles Forsyth
Vita Nuova

This work was supported in part by the Department of Energy Office of Science under award number DE-FG02-08ER25851

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Contents

This report documents the results of work done over a 6 year period under the FAST-OS programs. The first effort was called Right-Weight Kernels, (RWK) and was concerned with improving measurements of OS noise so it could be treated quantitatively; and evaluating the use of two operating systems, Linux and Plan 9, on HPC systems and determining how these operating systems needed to be extended or changed for HPC, while still retaining their general-purpose nature.

The second program, HARE, explored the creation of alternative runtime models, building on RWK. All of the HARE work was done on Plan 9. The HARE reseachers were mindful of the very good Linux and LWK work being done at other labs and saw no need to recreate it.

The organizations included LANL (RWK) and Sandia (RWK, HARE), as the PI moved to Sandia; IBM; Bell Labs; and Vita Nuova, as a subcontractor to Bell Labs. In any given year, the funding was suffcient to cover a PI from each organization part time.

Even given this limited funding, the two efforts had outsized impact:

- Helped Cray decide to use Linux, instead of a custom kernel, and provided the tools needed to make Linux perform well
- Created a successor operating system to Plan 9, NIX, which has been taken in by Bell Labs for further development
- Created a standard system measurement tool, Fixed Time Quantum or FTQ, which is widely used for measuring operating systems impact on applications
- Spurred the use of the 9p protocol in several organizations, including IBM
- Built software in use at many companies, including IBM, Cray, and Google
- Spurred the creation of alternative runtimes for use on HPC systems
- Demonstrated that, with proper modifications, a general purpose operating systems can provide communications up to 3 times as effective as user-level libraries

We describe details of these impacts in the following sections. The rest of this report is organized as follows: first, we describe commercial impact; next, we describe the FTQ benchmark and its impact in more detail; operating systems and runtime research follows; we discuss infrastructure software; and close with a description of the new NIX operating system, future work, and conclusions.

Chapter 1

Commercial Impact

1.1 Cray

In August 2006, we had meetings with Cray to explore the use of Linux on their supercomputers in place of the Light Weight Kernel then shipping. The discussions aligned with the basic thesis of our Right Weight Kernel project, which was that a properly designed subset of a commercial kernel, coupled with a set of carefully designed extensions, could compete with Light Weight Kernels, in additon to bringing in the advantages of a more standard programming environment.

Larry Kaplan, of Cray, notes: “The FastOS forums sponsored by the DOE Office of Science were a very important part of Cray’s investigations into compute node operating systems for HPC. The Right Weight Kernel project in particular provided important information on the challenges of using Linux which helped drive the research and development done at Cray in order to deliver the Cray Linux Environment. Cray fully believes in the importance of the research community and the ability for them to help influence our direction and success and looks forward to future opportunities for collaboration.”

As part of Right Weight Kernel project, we created the Fixed Time Quantum[19] benchmark in order to support quantitative evaluation of OS noise. Cray used this benchmark to evaluate and improve the performance of their Compute Node Linux product[11].

1.2 IBM

IBM Research’s involvement in this project has been extremely valuable in exploring ways to broaden the applicability of the BlueGene hardware through alternative software stacks. It has allowed us to explore hardware features which aren’t used by the production software as well as explore alternative schemes for scalability and reliability which are difficult or impossible under the production software. This alternative view has given us new insights which can then be incorporated into future hardware designs as well as future production system software.

IBM also used FTQ to instrument the Blue Gene kernel[6][15]; LLNL used FTQ to measure a performance problem on the Purple machine, and discoverted a previously unknown impact of the Power Hypervisor.

1.2.1 9p

Our involvement in the HARE project has increased the visibility and use of the Plan 9 resource sharing protocol, known as 9p, within the broader IBM community. In particular, during the course of the HARE project we incorporated both 9p and the XCPU workload management system into a hybrid computing platform called the Virtual Power Environment, which was developed for the Mare Nostrum supercomputer at the Barcelona Supercomputing Center. Infrastructures derrived from the original VPE idea are still under development by HPC Links under the product named VERTEX [16]. Experiences from the VPE effort were factored back into XCPU resulting in the creation of the XCPU2 [8] workload mangaement framework which then was factored back into the HARE project as XCPU3 [18] and the Unified Execution Model [21]. The fundamental components, Linux kernel support, and advanced XCPU models were all developed as a direct result of the HARE funding.

Outside of the high performance computing area, the 9p protocol has achieved great success within the emerging virtualization and cloud computing area as the defacto cloud computing paravirtualized file system under Linux. In coordination with IBM Research, development teams have created a virtio [17] transport for 9p and incorporated a 9p server into Qemu [3] which allows any KVM machines to be able to directly mount file systems from the host [10]. This technology has been part of the main line kernel for about a year and is being incorporated into the commercial Linux distributions.

1.2.2 Hybrid Kernels

Additionally, our experiences with hybrid environments on BlueGene with different kernel types running on different nodes has led us towards a hybrid kernel model for future massive multicore platforms where we are looking at taking advantage of running different types of kernels on different cores in order to improve performance and efficiency by insulating application cores from operating system noise. These so called ”hybrid” kernels are currently being evaluated to be the default production kernel on future platforms.

1.2.3 Analytics

Several of the workload distribution methodologies we explored during the HARE project are also being evaluated for use in large-scale analytics systems. In particular, the unified execution model and PUSH [5] workflow are being looked at for application within analytics clusters as an alternative to Hadoop for certain scenarios and the multipipe technology developed as part of HARE is also being evaluated for integration into several internal projects.

In addition to workflow management, IBM is also exploring the application of 9p synthetic and static file system technologies to analytics file systems which can be used with our execution model or with more conventional analytics software such as Hadoop.

1.3 Coraid

Coraid is a fast-growing vendor of network disk systems used in cloud computing centers. Coraid uses the trace software we developed[14] for Blue Gene. This software extends both the kernel and the linker to enable convenient, efficient performance tracing of kernel functions. A measure of the efficiency of this software is that all of Coraid's production kernels ship with the tracing software always installed and enabled. The capability provided by the software is so useful, and the performance impact so small, that Coraid decided that it should always be available.

1.4 Google

As a first step in the FAST-OS project, in 2005 we funded the creation of a compiler toolchain (C compiler, linker, and assembler) for Plan 9 on the AMD/Intel 64-bit architectures, known as Opteron and EM64-T. As per a basic requirement of the project, these tools were created as open source.

Google has recently released a new programming language called Go. Go supports, among other architectures, the AMD/Intel 64-bit CPUs. For these CPUs, Google adapted the Plan 9 C toolchain we created. There is hence a very direct link from the FAST-OS funding to Google's Go programming language, a language which is seeing very wide adoption.

Chapter 2

Fixed Time Quantum

Fixed Time Quantum[19] is a tool for creating measurements of operating system noise, allowing users to do quantitative analysis with standard tools such as Matlab and GNU Octave.

FTQ has found wide use in both the research and commercial world, having become one of the standard measurement tools for OS noise. Both IBM and Cray have used it to help tune their operating system kernels for minimal noise. FTQ results supported Cray's decision to move to Compute Node Linux, replacing the Catamount Light Weight Kernel originally delivered with those systems. Real application results supported the initial results from FTQ.

The distinguishing feature of FTQ is that it measures work per fixed amount of time, rather than the more traditional time taken to do a fixed amount of work. The distinction is subtle but essential. FTQ is written to carefully ensure that the time samples are for a fixed amount but also that the sampling interval is *stationary.* FTQ ensures this property by starting each sample interval at the correct point in team even if, due to interference, the previous sample interval took too long. For more discussion of how this is done the reader is referred to the original paper.

As mentioned, FTQ allows users and vendors to use quantitative, rather than qualitative analysis. As an example we will walk through real data taken on Blue Gene/P on a very early implementation of Plan 9. The FTQ program produces two output files: a so-called "counts" file, and a times file. These two files allow a user to reconstruct how much work was done per interval. The files are useful independent of each other: the counts data gives some idea of how much work was done, so that the same binary, under different operating system, can compare how well the applications run. The times data can be used to determine how well the algorithm that maintains stationary sampling is doing its job; deviations of the time from a desired baseline can point to operating system, virtual machine, or even hardware problems: the times file was used to diagnose virtual machine interference on the Purple system at LLNL. Combined, the two files can be used for deeper analysis, combining both work and time information.

While the raw data is useful, it is possible to misread too much into a raw data plot. In the original paper we showed two traces, one of them generated by white noise and the other by a very well-defined signal. To the naked eye, they are indistinguishable. Further processing of the data is essential to ensure that the measured data represents information, not white noise.

In Figure 2.1, we show a similar plot for three FTQ runs, on CNK, Linux, and Plan 9. What is of interest are not the actual numbers – the Plan 9 binary is a different binary – but the occurrence of spikes in the graph. One can gain some information immediately: there is clearly periodic interference of the same frequency for each operating system. Further investigation revealed that the 64-bit processor clock, comprised of two 32-bit counters, had a glitch when the low-order 32-bit counter rolled over, leading to a slightly longer sample interval at that point. The algorithm quickly resynchronized, however, such that the counts over a long period were correct. This spike can easily be misinterpreted in the time domain graph; it does not have any impact on the frequency domain graph.

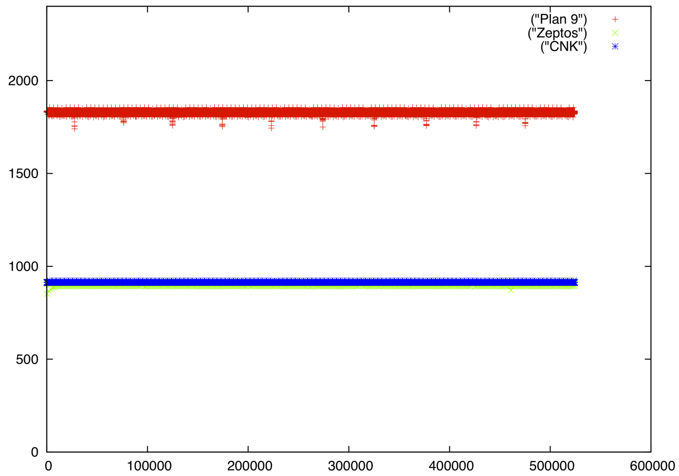


Figure 2.1: **A plot of FTQ 'count' data for three Blue Gene/P operating systems, with sample number on the X axis and unit-less work on the Y axis.**

Converting this data to the fequency domain is relatively straightforward. We present the octave script in Figure 2.2. The program reads the file in using a function that returns a one-dimensional vector. The function `fitmax` normalizes all the values in the vector to the maximum value. Finally, the code calls the octave pwelch function, which generates two vectors: a power spectrum

```
samples = readcounts('9ftq63_0_counts.dat');
samples = fitmax(samples);
[Psd,w] = pwelch(samples,[],[],[],810);
```

Figure 2.2: Octave script for processing raw FTQ data. We only use the counts file in this case.

estimation (Psd) and a set of frequencies(w). The parameters are the samples, and the cycle counter frequency (810 Mhz.). We show a plot of the Power spectral density for the three operating systems in Figure 2.3

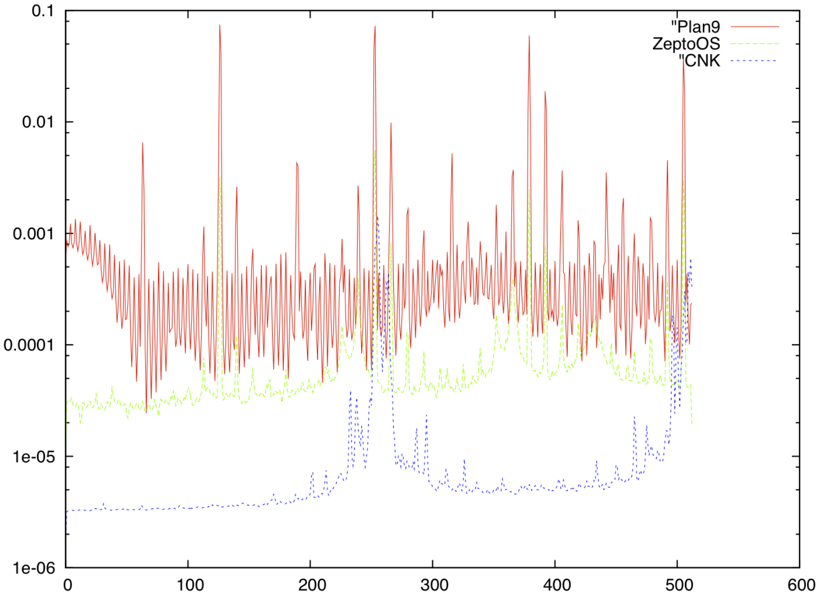


Figure 2.3: **Power Spectral Density for the three operating systems, Frequency in HZ on the x axis and unit-less amplitude on the Y axis.**

The graph shows that CNK has a better noise figure than either of its two more general purpose competitors. It is also possible to see the frequencies at which noise spikes occur. In fact, an earlier version of this chart revealed a very distinct noise spike at 30 HZ. which we eliminated. Finally, it is easy to see that, while there is an apparent similarity in the time domain between CNK and ZeptoOS noise,they are not at all similar in the frequency domain. One can not simply look at raw data graphs and reveal all the hidden information.

FTQ allows us to quantitatively measure and analyze noise, using standard signal processing techniques. Spectral components of noise can be measured, traced back to their source, and eliminated.

Chapter 3

Operating Systems Developments

3.1 Plan 9 Port to Blue Gene /L and /P

3.1.1 Large Pages

Blue Gene processors use software-loaded TLBs. A TLB manages the virtual to physical mapping for a single page, which in both Linux and Plan 9 is 4096 bytes. The processors support more than just one page size, in multiples of 4, from 1024 bytes to 1 Gbyte (with a few sizes missing). The CNK features 1 Mbyte TLBs, and benefits from reduced TLB fault interrupts.

We extended Plan 9 to support larger TLBs. In order to measure the improvement we used the strid3 benchmark to show the impact of TLB size upon an application. Shown in Figure 3.1 is the result. Plan 9 is eight times slower than the CNK when 4096 byte pages are used. With one Mbyte pages in Plan 9, its performance is identical to the CNK. The change to Plan 9 to support the larger pages amounted to less than 30 lines of code[12].

3.2 Currying and Process-private system calls

Extensive measurement of the kernel, using the tool developed and described in [14], showed that for read and write system calls a great deal of overhead was repeated for each iteration of the call. Two of the biggest overheads are validation of the file descriptor and the read/write pointer and length validation. We also found in most cases that the pointers are being reused for each call: it is extremely rare for these parameters to be wrong, yet programs pay the full overhead for checking them each time. For the Blue Gene global barrier device, the overhead for the system call was 3.529 microseconds.

In [13] we describe the possible approaches to resolving this problem. We developed two several extensions to Plan 9 that allowed the kernel to Curry the

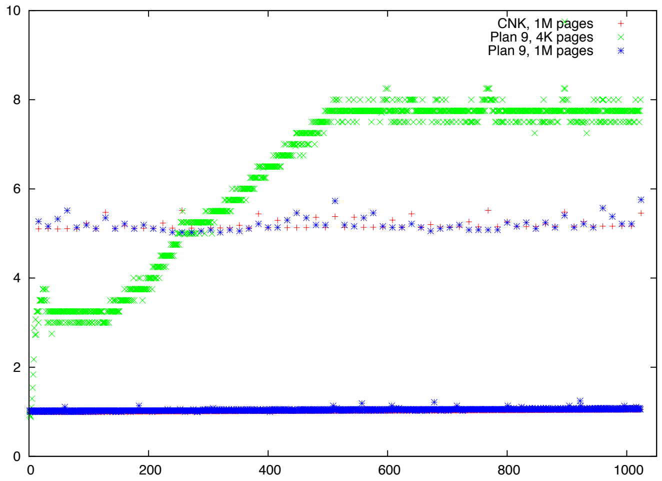


Figure 3.1: **Result of running strid3 on CNK, Plan 9, and Plan 9 with 1 MByte pages.**

arguments to the read and write system calls. The result: the file descriptor and pointer/length information is checked once, and the result stored in a cache held in the proc struct (per-process information) in the kernel. The per-process structure is also extended with a private system call table. When the process makes a private system call, all the standard checking is skipped and the pre-validated cached values are used instead. This new system call path resulted in a 5-fold reduction in overhead, to 729 nanoseconds.

The Currying and per-process system calls together form a new type of interface to operating system kernels.

3.2.1 Shared Heap

Currying reduces the overhead for process-driven operations. It does nothing for interrupt-driven operations, because interrupts, by design, do not occur in a process context. Page pinning is used to resolve this problem on Linux systems: process memory mappings are communicated to the driver and "pinned" into place. Page pinning adds a great deal of code to a kernel; in some cases the pinning code for (e.g.) Linux is larger than all of Plan 9 itself.

We implemented a "shared heap" model in Plan 9. All process heaps, the kernel, and the drivers share a common address space, consisting of the top 1.5 Gbytes of memory. All heap (not text, stack, or data, just heap) is allocated from this shared 1.5 Gbytes of memory. A given heap address in a process, kernel, or interrupt handler will point to the same piece of memory. In contrast, in a standard OS, the same heap address in each process points to different memory. Note that a shared heap address space does not imply that all the data in the heap itself is shared between all the processes. Processes can share parts of the heap, but are typically limited to their own private piece, and can not see other processes heap data.

Processes can easily pass pointers to each other because a given heap pointer has the same meaning in every process. Processes can communicate an address to the driver for use by the interrupt handler and there is no need for complex pinning, because the address has the same meaning in every machine context.

We further modified the BG/P Torus driver to implement queues on the receive side. As packets are received by the interrupt handler they are directly moved to the queues. [1] Finally, once the driver and kernel were extended, we created a library to allow processes to send from the shared heap and receive into queues in the shared heap.

Our resulting performance was better than that of the MPI library, even while using the kernel for I/O. In fact a simple ping-pong code ran with about 1/3 the latency of MPI. Our code was about 100 times smaller than the MPI library. Further, the device could be made to work for multiple independent applications, which is not possible on BG/P hardware if the direct-access MPI libraries are used. We thus showed that if a kernel provides the proper abstractions, it is not necessary to run the network device driver in user memory, as is done today.

3.3 File Systems

3.3.1 Compute Node Caches

We needed to reduce the load on the file server shared by thousands of nodes in a Blue Gene Plan 9 cluster. The intent was to execute the caching mechanisms on the I/O nodes as well as look at spreading them out amongst the compute nodes depending on the I/O requirements of a particular application.

Plan 9 had two existing caching mechanisms for 9P file servers: a volatile cache controlled by devmnt, and a persistent disk-based cache managed by the user-level server cfs. Both sent the server all 9P requests except reads satisfied by the cache. Cfs was quickly converted to a ramcfs that used the large memory of a Blue Gene I/O node instead of a disk, but most 9P traffic still passed through. A redesign produced fscfs, which breaks the direct link between its client's 9P transactions and those it makes to the server, producing a dramatic reduction in traffic seen by the server.

Fscfs is a normal, user-level C application, using the Plan 9 thread library, libthread, for internal concurrency. Like cfs, it is a 9P transducer: it acts as a client to a remote 9P file server on one connection, and acts as a 9P server

[1] These queues are very similar to those provided on the T3D. In the T3D, however, they were implemented in hardware.

to its client on another connection. (Once that connection is mounted in the name space, many client processes can share it.) To provide data caching, fscfs uses the same strategy as the existing caches: it makes a copy of the data as it passes through in either direction, and stores it in a local cache. Subsequent read requests for the same data will be satisfied from the cache. Write requests are passed through to the server, replacing data in the cache if successful. Cached data is associated with each active file, but the memory it occupies is managed on a least-recently-used basis across the whole set of files. When a specified threshold for the cache has been reached, the oldest cached data is discarded.

Fscfs maintains a path tree representing all the paths walked in that tree, successfully or unsuccessfully. A successful walk results in an end-point that records a handle referring to that point in the server's hierarchy. (Note that intermediate names need not have server handles.) If a walk from a given point failed at some stage, that is noted by a special path value at that point in the tree, which gives the error string explaining why the walk failed. If a subsequent walk from the client retraces an existing path, fscfs can send the appropriate response itself, including failures and walks that were only partially successful. If names remain in the walk request after traversing the existing path, fscfs allocates a new handle for the new path end-point, sends the whole request to the server, and updates the path appropriately from the server's response. Remembering failures is a great help when, for instance, many processes on many nodes are enumerating possible names for different possible versions of (say) Python library files or shared libraries, most of which do not exist. (It would be more rational to change the soft ware not to do pointless searches, but it is not always possible to change components from standard distributions.)

Fscfs is just over 2,000 lines of C code, including some intended for future use. It has more than satisfied our initial requirements, although much more can be done. It aggregates common operations in a general but straightforward way. Its path cache is similar to Matuszekps file handle cache in his NFS cache, and closer to 9P home, some of the subtle cases in handles management in fscfs seem to turn up in the implementation of 9P in the Linux kernel, where Linux lacks anything corresponding exactly to a 9P handle.

3.4 New network communications models

3.4.1 Active Message Support

As a performance measurement experiment we implemented Active Messages in the Torus driver. Our implementation of shared heaps, mentioned above, allowed us to do the active message operation in the torus interrupt handler. The result was an ability to perform remote put/get/queue insert at a rate roughly three times greater than when MPI was used.

3.4.2 High performance network transport

3.4.3 Network programming models

Writing or converting applications to make best use of runtimes such as MPI can be non-trivial: to avoid the synchronization imposed by matching sends and receives, and barriers, and to overlap communication to reduce latency, programmers use non-blocking primitives and asynchronous communication.

In UNIX, a program can read and write on a file descriptor to communicate with a local device, or a TCP/IP port on a distant machine, without changing the program text or even recompiling it: IO through file descriptors is location independent. The intention behind developing specialized protocols—the bulk messaging protocol in particular—and the new messaging kernel interface is to allow HPC applications to be written in a similar style, but with efficient transport.

With IO through file descriptors, if two communicating processes are on the same node, the system will use cross-domain references to transfer the data directly; if they are on different nodes, the sender's data will be passed by reference through the kernel to the bulk messaging drivers, which will use DMA to transmit it to memory on the remote node, which will be passed by reference from the kernel to the receiving process. File descriptors hide the protocols used, not just the location. When debugging on a laptop, pipes can replace the fancy networks, without changing the program, or the communication semantics. It is at least as easy to use as the naive use of MPI's send/receive, but without imposing a static process structure, since new processes can be created on demand, and added to the communication set, in the same way as in network applications.

The primitives are usable both directly by applications and by supporting services. For example, we ported to Plan 9 a small scientific application that used MPI, in which asynchronous communication had resulted in a complex structure. We replaced all MPI calls by calls to a small library using our primitives. (The library provides *reduce* and *allreduce* operations, for instance.) The revised program is written in a straightforward style, like a simple network application, as if the communication were synchronous, but the kernel and its protocol drivers exploit the potential for concurrency and overlap. As another example, we have been experimenting with a small library and associated services that creates a distributed spanning tree (which can be made fault-tolerant) to provide multicast and convergecast to support large-scale computations.

3.4.4 Data network basics

The Blue Gene system provides several data networks. The IO nodes are linked by Ethernet, with connections through switches to the outside world, providing a conventional IP network. The CPU nodes are connected by the Torus; and the IO and CPU nodes are linked in a collective network. The only path from a CPU node to the outside (including external file service) is through the collective to an IO node, and then through the IP network on the Ethernet. The Ethernet supports large packets ('jumbo frames'). The collective and Torus, however, provide only small packets: 256 and 240 bytes of user data respectively. On BG/P, compared to BG/L, the Torus has extra DMA interfaces to the Torus that can automatically fragment messages into the network's tiny payloads, although the recipient must still reassemble messages.

The Plan 9 IP stack is completely independent of the MAC layers: a small *medium* driver connects the bottom of the IP stack to the files representing the driver for a given MAC level. For example, the Ether medium driver opens `/net/ether0/data`, reads and writes the file, converting as it does so between the Ether packets the device expects and IP packets, adding and removing MAC headers, and invoking ARP as required. It was a simple task to write specific medium drivers for each of the Torus and collective networks, just several hundred lines each. Note that unlike some other systems on Blue Gene, Plan 9 therefore does not emulate an Ethernet on either torus or collective to implement IP. Not only is the MAC-level framing of IP packets specific to each device, but there is no need for ARP (which historically has been a problem when attempting to scale to thousands of nodes).

Raw access (ie, non-IP) to each network is also provided, through specific device files in the name space, such as `/net/torusdata`; in fact, it is the same interface used by the IP medium drivers. Applications that use the networks outside IP often need to send messages larger than the hardware's payload. We therefore added a simple fragmentation protocol, allowing messages of up to 64 kbytes to be exchanged across the network. The implementation is about 300 lines, allowing for an implementation entirely in software on BG/L, and a hybrid implementation on BG/P that takes advantage on the Torus of support in the DMA subsystem for fragmentation, using a software header encoding that allows correct but efficient reassembly at the receiver even when several nodes are communicating with the same recipient. Note that combining fragmentation with IP is worthwhile: the basic IP header is 20 bytes, and TCP adds 20 more, which is a non-trivial overhead for the tiny Torus and collective packets. Fragmentation allows one TCP/IP header to be used for a large message, which is then fragmented; although there is overhead in fragment reassembly, the resulting message traverses the IP stack only once, instead of many packets.

3.4.5 Transport protocols

As noted elsewhere, Plan 9 uses the 9P protocol to implement a range of system and application services, including distributed services, such as those provided by the Unified Execution Model. Unusually for modern network protocols, 9P is not 9P/IP: it is not expressly an Internet protocol. It can use any underlying transport that provides reliable, sequential (in-order) delivery; the current version of 9P does not even require the transport to respect 9P message boundaries. Having implemented an IP network above both Torus and collective networks, and thus TCP/IP, we could immediately deploy 9P. The Blue Gene environment, however, like many supercomputer systems, makes it easy to experiment with alternative protocol designs and implementations. Although the nodes, directly or indirectly, access resources on outside networks such as file service or graphics display, much of the communication is intra- and inter-node. Furthermore, it is easy to change the protocol implementation at all nodes from run to run, simply as a by-product of rebooting the system. By contrast, there are many barriers to experimenting with new protocols on the Internet. For example, Plan 9 historically had a reliable datagram protocol, called IL/IP, tuned to the needs of 9P. It has seen reducing use because on the wider Internet, intervening routers and firewalls do not recognise the protocol; nor is it easy to change them to do so.

As well as supporting 9P, which is a reliable point-to-point conversation between client and server, we wanted to support reliable messaging between arbitrary pairs of processes (between any nodes). On Blue Gene, of course, the underlying hardware networks provide just such reliable delivery, at least in principle, and we had added fragmentation to allow larger, more expressive messages. On the other hand, in the larger context of the project, we were interested in fault-tolerance, including operation on unreliable or failing networks. Furthermore, because of its intended use for a single application using MPI, the hardware's properties are not quite right for 9P, or even general inter-process messaging. Most obviously, reliability and flow-control are between nodes, not between collections of processes on those nodes. Adding multiplexing of inter-node channels, for example by running IP, requires keeping the receiver 'live' (ie, unblocked) at all times, to prevent the hardware flow-control from triggering and causing deadlock. Thus, for reliable communication, the hardware level flow-control is neither necessary nor sufficient, and flow-control must be added somewhere in the multiplexing software stack.

Multiplexing, datagram and transport protocols are superficially straightforward, but typically become burdened with much practical detail, for connection establishment, error control, flow control, and congestion control. Rather than risk the time to develop a complete protocol from scratch, we decided to adopt and adapt an existing complete protocol definition, or more probably a suitable subset of one, limiting the implementation to just what we needed in the Blue Gene environment. We needed multiplexed messaging connections for 9P, and reliable but connectionless messaging for computational work. The first attempt was loosely based on the Xpress Transport Protocol,[20] which had the attraction of supporting unreliable and reliable datagrams, and reliable connections as supersets of each other. The protocol essentially microcodes the effects desired by the application level; thus, a reliable datagram includes one protocol header that declares a new connection, transfers data into it, requests an acknowledgement, and closes the connection; a reliable connection instead puts those operations in the headers of distinct messages, in particular leaving the connection open after the first message, and taking advantage of the context thus established to abbreviate subsequent messages. Plan 9's TCP/IP has 3236 lines for the TCP level, 706 for the IP level, and some auxiliary code. By contrast, the XTP prototype had 1740 lines of code, and needed no IP level, but supported not just TCP-like connections, but reliable datagrams as well.

Unfortunately, connection establishment and reset in XTP still seemed overly complex for this application, requiring message exchanges and timers. A second attempt was made (reusing much of the code), inspired by a much older protocol *delta-t*.[23] It was simpler than XTP, and had a unique, timer-based approach to connection management. Briefly, in delta-t a logical connection exists between every possible sender and every possible receiver, but a connection is quiescent—thus requiring no state on either sender or receiver—if no message has been sent or received for a specified interval. The delta-t designers observed that timers are required to detect time out, and to provide protection against lost messages on connection close, but the timers can be used instead to make the connection messages redundant. Thus, to use a connection, a process simply sends a message on it, and the receiver then creates the state for the connection. After an optional acknowledgement, if no further messages are received (or the receiving process has no reply), the connection becomes quiescent and the state is removed. In the original protocol there are therefore no extra messages for connection establishment or shutdown, and the protocol can carry a single message efficiently. An old quip is that one cannot afford to have (say) TCP/IP connections between every pair of 65,000 nodes. With the delta-t approach, the pair-wise connections (logically) exist but cost nothing, until messages are actually sent, when the transient state can carry as much data as needed, reliably, and without extra messaging for connection establishment.

On the other hand, and currently the primary use, a delta-t connection can carry 9P traffic between a client and a 9P server. Because the interface to the protocol (ie, `/net/pk`) obeys the Plan 9 name space conventions for networks, applications can use the protocol instead of TCP/IP without change.

3.4.6 Bulk messaging

On BG/L, the Torus network hardware presented a collection of memory-mapped FIFOs as its interface, accessed from the processor by explicit loads and stores. The BG/P Torus hardware hides that detail behind a collection of rings of DMA descriptors, and new DMA hardware moves the data between memory and selected FIFOs. The hardware goes much further, however, which required revisiting the software network interfaces to the Torus. (The collective networks were unchanged.) In particular, the hardware provided new DMA operations: *put* and *remote get*, implemented by special packets interpreted by the DMA engine. The Put operation causes the hardware to send an arbitrarily-large block of memory from one node, across the Torus, directly into a block of memory on a specified target node. The destination address is specified as an offset from a base address in a table on the remote node, selected by a 'counter ID' specified by the sender. In the other direction, one node sends a Remote Get packet to a second node, causing its DMA engine to execute a *Put* operation. Usually the target of the Put is the first node, so that the original message thus gets the relevant data from the second node efficiently using DMA.

To exploit the BG/P extensions, the Plan 9 Torus driver was extended to implement a lightweight protocol for bulk messaging. A process wishing to use bulk messaging opens the file `/net/torusbig`, and sends and receives messages using write and read system calls. The first 32 bytes of the data in each read or write is a software header giving source and destination network addresses, an operation code, a parameter, and 24 bytes of uninterpreted data for use by the application. That is followed by the data of the bulk message itself. The uninterpreted data might contain values for subaddressing, or metadata describing the data, but that is up to the application. The operation code and its parameter are reserved to the protocol subsystem.

Bulk messaging is a network protocol. Writes and reads are not matched, let alone synchronised, even in the implementation of the protocol. A write causes a bulk message to be sent to the destination node, which is queued until some reader does a read. The application data attached to the message might be used to associate requests and replies in an application level algorithm, perhaps by library code, but that is outside the messaging protocol.

In the Torus driver, the protocol implementation uses the normal Torus packet channels, but bulk message control packets are specially marked, and interpreted by the interrupt handler on the destination node. A bulk write request sends a Get control message to the destination. The control message includes the metadata from the request, and addressing information for the data. If the destination cannot process the request, it will reply with a Nak. Otherwise, it will issue a DMA Remote Get request, quoting the addressing information it was given. The hardware on both source and destination nodes is programmed to signal an interrupt when the transfer completes. The sending node marks the bulk request done; the receiver queues the data to be read. Control packets carry tags to allow requests to be cancelled, acknowledged, or nak'd if required.

Note that the bulk writer is not required to wait for the transmission to complete, allowing the usual overlap provided by buffering. More subtly, although the transfer is logically a 'put' from source to destination, the *receiving* kernel initiates the transfer using a DMA Remote Get. There are several reasons. First, the parameters for a DMA Put require the sender to specify the correct index in a table on the *receiver*, and that can only be known by global convention or prior arrangement (ie, through a previous exchange of messages), neither of which supports dynamic communication well. Second, the receiver needs to find the memory to receive the data, and might be receiving from many other nodes. Having the receiver start the transfer allows various forms of flow control. Most important, however, using DMA Remote Get is more robust against node failure. If the sending node fails and stops, the request will fail or time out. If the sending node fails and restarts, at worst the Get request will fetch incorrect data, or receive an error. The metadata can be used to check the data. By contrast, if a DMA Put is used, and a destination node fails and restarts, it is possible that, without an error being notified, the put will transfer into whatever memory is addressed at the time by the table index parameter, corrupting the target node.

3.4.7 IO segments

For initial experiment, we limited messages to 128 KiB, because of addressing constraints in the system. We did, however, want bulk messaging to be able to send much larger chunks of data. Furthermore, the normal read and write system calls imply copying, which is obviously inefficient. We changed the system to allow both restrictions to be removed, by solving a more general problem. We added a simple form of cross-domain reference.

A process can attach a special type of segment intended to be used for IO. The segment is backed by physically-contiguous memory, usable by hardware DMA. Inspired by the x-kernel [7], fbufs [4], and System 360's LOCATE mode, two new system calls were added:

Message readmsg(int fd, uint iolimit)
int writemsg(int fd, Message m)

Like the read system call, readmsg fetches data from the file denoted by file descriptor fd, but instead of having the user provide a buffer address, readmsg returns a Message value that refers to the data read (ie, the buffer address), and any associated metadata (eg, source address). Similarly, writemsg accepts a Message value instead of a single buffer address. The data address in the Message can be an ordinary address in the process. but it can also be a reference to an IO segment. Such references can be passed between user process and device driver through the kernel. There are primitives to manage the memory in an IO segment.

In the HARE implementation on BG/P, IO segments and the system calls are restricted to use with particular devices, specifically the Torus. Since then, however, we have revised and extended the scheme in the context of the 64-bit NIX work, to handle IO to arbitrary devices, and support interprocess communication.

3.5 Kittyhawk Kernels and VMM

Supercomputers and clouds both strive to make a large number of computing cores available for computation. However, 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. Project Kittyhawk [2] was an undertaking at IBM Research to explore the use of a hybrid supercomputer software infrastructure on top of a BlueGene/P which allows direct hardware access to the communication hardware for the necessary components while providing the standard elastic cloud infrastructure for other components.

Beyond the enabling infrastructure for cloud provisioning and dynamic boot of multiple images across compute nodes within a single BG/P allocation, the Kittyhawk team also ported a Linux compute node implementation complete with an Ethernet interface to the high-performance collective and torus networks [1]. This allowed Linux applications to run unmodified on Blue Gene compute nodes without incurring the performance overhead of reflecting sockets and I/O operations thorugh user space helpers as in ZeptoOS. It also enabled more conventional storage solutions including SAN and NAS to be extended into the compute node network for I/O intensive solutions. While the Kittyhawk Linux kernel provided this Ethernet interface, it also provided an API allowing the allocation of torus channels directly to high performance applications which knew how to interact with them allowing even greater degrees of performance and flexibility. These allocated application channels co-existed with the kernel-space channels providing a good hybrid trade off between system provided networking and OS bypass.

The Kittyhawk environment was not limited to Linux kernel targets. It also supported execution of the L4 microkernel operating system. In addition to providing an alternative execution platform for applications, the L4 Kittyhawk port also supported a virtual machine monitor which could be used to further virtualize the BG/P nodes and network providing additional protection in cloud deployment environments.

The Kittyhawk infrastructure, the linux kernel patches, and the L4 kernel and virtual machine monitor have all been released as open source and are available from http://kittyhawk.bu.edu.

Chapter 4

Run Time Developments

Developing runtimes for processes on Plan 9 was challenging, because we did not opt for the traditional HPC approach of placing device drivers in user level libraries. From the beginning of this project, we had decided to have the kernel control communications and resource management tasks that have been relegated to user level runtimes for decades. Returning control of these tasks to the kernel makes the HPC resource available to a much wider set of applications than just MPI programs. On many traditional HPC systems, high performance networks are not even visible to non-MPI applications. On Plan 9 on BG/P, even the command shell has access to the fastest networks: hence it is possible to run a shell pipeline between compute nodes, without rewriting all the programs involved to use MPI. Parallel programs become more portable, and much easier to write and use.

Removing device drivers from libraries removed all the attendant complexity from those libraries; many user-level libraries have more complex code than the Plan 9 kernel, and even the smallest of those libraries is larger than our kernel. Our limited-function MPI supports many MPI applications in less than 5000 lines.

User-level runtimes are not without their advantages. Most MPI libraries are thread-safe, and incorporate an activity scheduler that can choose activities with very low overhead. Because the device driver is contained in the process, pointers to data are valid for all the operations the runtime performs, including setting up DMA. There is no need to translate addresses, and on a simple system like Blue Gene, there is no need for complex page pinning software. These MPI runtimes in essence create a single-address-space operating system, complete with drivers and scheduling, that is "booted" by the host operating system and then left to run in control of the node. Again, the advantages of the runtime include the implementation of system call functionality with function calls; and a single address space, with the attendant elimination of address translations and page faults and elimination of unnecessary copies.

Leaving device drivers in the kernel requires reduced overhead in some areas. A system call takes 1-2 microseconds, on average; this time must be reduced when latency-sensitive operations take place. User-level addresses must be mapped into the kernel address space. In interrupt handlers, user-level memory is not easily accessed in Plan 9, since interrupt handlers have no user context.

In this section we describe the research we have undertaken to address these needs.

4.1 Unified Execution Model

In order to address the need for a more dynamic model of interacting with large scale machines, we built the unified execution model (UEM) [21] which combined interfaces for logical provisioning and distributed command execution with integrated mechanisms for establishing and maintaining communication, synchronization, and control. The intent was to provide an environment which would allow the end-user to treat the supercomputer as a seamless extension of his desktop workstation – making elements of his local environment and file system available on the supercomputer as well as providing interfaces for interactive control and management of processes running on top of the high-performance system.

Our initial attempt at providing the UEM infrastructure was based around an extension of some of our previous work in cluster workload management [9] [8]. This initial infrastructure, named XCPU[3] was built on top of the Inferno virtual machine which could be hosted on a variety of platforms (Windows, Linux, Mac, Plan 9), while providing secure communication channels and dynamic name space management. It functioned as a server for the purposes of providing access to the elements of the end-users desktop as well as an gateway to the interfaces for job control and management on the supercomputer. The management interface was structured as a synthetic file system built into the Inferno virtual machine, which could be accessed by applications running on the host either directly via a library or by mounting a 9P exported name space (using the Linux v9fs 9P client support built into the Linux kernel).

A key aspect which allowed the UEM to scale was a hierarchical organization of node aggregation which on bluegene matched the topology of the collective network. It became clear that at the core of the interaction of the UEM infrastructure were one to many communication pipelines which were used to broadcast control messages and aggregate response communication and reporting. We extracted that key functionality into a new system primitive called a multipipe [22]. The result dramatically simplified the infrastructure and improved overall system performance. It also became clear that multipipes were a useful primitive for the construction of applications and other system services. We extended the design to incorperate many to many communication as well as support for collective operations and barriers.

While Inferno provided a great environment on top of end-user desktops, we didn't want to incur the overhead of running Inferno on all the compute nodes of the supercomputer. Using multipipes as the core primative, we built a pair of synthetic file systems for Plan 9 which provided an interface for initiating execution on a node named execfs and a second file system whose purpose was to coordinate groups of remote processes to allow for fan-out computation and aggregate control named gangfs. The gangfs file systems were built to organize themselves hierarchically in a fashion similar to XCPU[3]. This could then be mounted by front-end systems or end user workstations and interacted with using Inferno as a gateway and server.

In order to make interactions with the UEM file system a bit more straightforward we developed a new shell which incorporated support for multi-pipes and interactions with the taskfs and gangfs interfaces. Push is a dataflow shell which uses two new pipe operators, fan out(’—¡’) and fanin(’¿—’) to create Directed Acyclic Graphs of processes. This allows construction of complicated computational workflows in the same fashion one would create UNIX shell script pipelines.

Chapter 5

Infrastructure Developments

5.1 Blue Gene Debug File System

The Blue Gene incorporates a management network with JTAG level access to every core on the system which is uses for loading, monitoring and control. Our Plan 9 infrastructure represents the external interface to this JTAG infrastructure as a file system. This file system is served by a special application on an external system that connects to a management port on the service node which accepts JTAG and system management commands. Directory hierarchies are used to organize racks, midplanes, node cards, nodes, and cores. Each core has its own subdirectory with a set of files rep resenting the core's state and control interfaces.

This has the effect of representing the entire machineś state as well as special files for interacting with per-node consoles as a single file hierarchy with leaf nodes reminicent of the the /proc file system from Linux (except more powerful). The monitoring network and debug file system were vital contributors to the speed with which we were able to bring up the system. Since all nodes and state are represented, it eases the job of writing parallel debuggers and profilers. Indeed, the Plan 9 debuggers are already written to work against such file systems and so this allowed us to debug multi-node kernelcode as easily as we would have debugged a multi-process applications.

5.2 nompirun

While the debug file system provided us a great environment for bring-up, it did not play well with the existing management infrastructures in production environments. To accomodate production environments we wrote management daemons for our IO nodes which interacted with the production management infrastructure for status reporting, parameter passing, and notification of termination of applications. We also constructed a number of scripts for interacting with the cobalt job submission system used at Argonne National Labs which automated launch of Plan 9 jobs including configuration of back-links to the front-end file servers and reporting channels for the aggreagted standard output and standard error of the jobs running on all the compute nodes. This alternate infrastructure was critical to allowing us to bring up applications with alternate (non-MPI-based) runtimes while still playing nicely with the production management software and job schedulers and was reused to enable the Kittyhawk profiles.

5.3 Kittyhawk u-boot and cloud provisioning

The production Blue Gene control system only allows booting a single image on all nodes of a block. Initial firmware which initializes the hardware is loaded via the control network into each node of the machine.

For Kittyhawk profiles, we take over the node from the firmware via a generic boot loader which is placed on all nodes by the existing control system. The boot loader, U-Boot, 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. Users can then use simple command line tools to request a node or a group of nodes from the free pool and push kernels to the newly allocated group. The kittyhawk infrastructure allows users to construct powerful scripted environments and bootstrap large numbers of nodes in a controlled, flexible, and secure way. Simple extensions to this model can also be used to reclaim and restart nodes from failure or simply to reprovision them with different base software.

Chapter 6

NIX

NIX is an operating system designed for manycore CPUs in which not all cores are capable of running an operating system. Examples of such systems abound, most recently in the various GPU systems. While most heterogeneous systems treat the non-OS cores as a subordinate system, in NIX they are treated as peers (or even superiors) of the OS cores. Many additional features of NIX were created based on what we learned from the RWK and HARE projects.

NIX features a heterogeneous CPU model and can use a shared address space if it is available. NIX partitions cores by function: Timesharing Cores, or TCs; Application Cores, or ACs; and Kernel Cores, or KCs. One or more TC runs traditional applications. KCs are optional, running kernel functions on demand. ACs are also optional, devoted to running an application with no interrupts; not even clock interrupts. Unlike traditional kernels, functions are not static: the number of TCs, KCs, and ACs can change as needed. Also unlike traditional systems, applications can communicate by sending messages to the TC kernel, instead of system call traps. These messages are ”active” taking advantage of the shared-memory nature of manycore CPUs to pass pointers to data and code to coordinate cores.

NIX has transparent support for a near-unlimited range of page sizes. On the 64-bit x86 family, for example, NIX supports 2 MiB and 1 GiB pages. Support for 1 GiB pages is crucial to performance: on one popular graph application we measured a 2x speedup when 1 GiB pages were used instead of 2 MiB pages. The implementation allows for virtual pages, e.g., one could provide 16KiB pages that are a concatenation of physical 4 KiB pages. These virtual pages would allow us to reduce the number of page faults an application causes, although they would not reduce TLB pressure.

NIX gives us the best of many worlds. Applications can own a core, and will not even be preempted by a clock tick, as they are even on Light Weight Kernels today. At the same time, the full capability of a general purpose operating system is available on core nearby. We feel that NIX represents the shape of future operating systems on manycore systems that are being built now.

Further information on NIX can be found in Appendis A.

Chapter 7

Future Work

This research has suggested a number of new directions for HPC.

- New operating systems that step outside the stale Light-Weight Kernel/General Kernel argument. We have shown that one such kernel, NIX, can support the desirable attributes of both and even outperform LWKs at the things they do best.

- Network IO can and should be done in the kernel, and future research should work out the API. OS Bypass should be bypassed.

- Network software that does not assume a perfect network needs to be created and put into use. This change in turn will allow innovation to occur in HPC hardware.

- HPC hardware needs to change so that it can provide near-perfect information about failures, but not nodes that never fail. Software can handle faults in a far more flexible manner than hardware.

- File systems must change to make explicit use of hierarchy. The ratio of Compute Nodes to IO Nodes should not be statically defined by wires, as it is today, but defined by the needs of the application, and hence flexible.

Chapter 8

Conclusions

While no commercial HPC vendor is using Plan 9, they are using much of the other software we developed, including the 64-bit compiler toolchain, FTQ, and NIX. There are other lessons learned:

- OS bypass is not required for good network performance. “Conventional knowledge bypass” is much more important: if we can get runtime libraries to accomodate such concepts as a shared heap address space, we can reduce runtime and OS complexity and remove OS bypass for the most part.

- The idea of quantitative measurement of OS noise was controversial when we first proposed it in 2004. In fact, there is a wealth of signal processing software and knowledge that can be applied to HPC. We need to move beyond qualitative descriptions to quantitative measurements. Adjectives should be avoided when numbers can be supplied.

- Users want a Unix-like API, even on systems like Blue Gene which do not run Unix on all nodes. Key requirements such as sockets, dynamic libraries, and a reasonably standard file system interface can not be waved away. Light Weight Kernels that fail to provide these services will either change (as did the CNK) or be abandoned by the vendor (as was Catamount).

- As pointed out above, “Thus, for reliable communication, the hardware level flow-control is neither necessary nor sufficient, and flow-control must be added somewhere in the multiplexing software stack.” Communications networks on future architectures could be made simpler, faster, more reliable, and lower power by taking this fact into account.

- We need to move beyond simply gluing hardware interfaces into existing kernel designs in non-optimal ways. One of the worst examples of this tendency can be seen in the Linux drivers for HPC networks: they all emulate an Ethernet. This emulation results in ineffeciencies and poor software structure: the Jaguar system, for example, has 30,000 hardwired

ARP-table entries *on each node* so that the TCP/IP address in $[x, y, z]$ form can be mapped to an Ethernet address in $[x, y, z]$ form. Ethernet interfaces are for Ethernet networks. There is no need to have a network interface for an HPC network emulate an Ethernet, as it does on Compute Node Linux on the Cray or ZeptoOS on BG/P.

This work was done as part of the FAST-OS program and its successor. The ground was prepared for the FAST-OS programs in a series of meetings held in the eary 2000s. One of the PIs earliest talks dates from 2002. In 2003, we made the following arguments:

- Linux is fun and working now; enjoy it
- Linux is not forever
- Some fundamental properties of the Unix model have fatal flaws
- We are entering a period of architecture flux
- We need diversity in OS research now to prepare for new, strange machines
- DOE can succeed in creating a successful, diverse OS research community

We might ask: is any of this less true now that our community is faced with the challenge of scaling parallelism up 1000-fold? We would argue that the problem is even worse. While this program may have had successes, in some sense the community is still plowing the same ruts. The extant machines still largely run Linux, and the problems we predicted in 2003 (for the 2012 timeframe) are starting to come to pass: with each new version, Linux gets just a bit slower, as it grows in size and complexity and the processors do not get any faster. As we stated in 2003, "At some point, like all other OSes, it is [Linux] going to fall over". Several large companies, in private conversation, have told the PI that one of the biggest management headaches they have is dealing with the increasing complexity of the Linux kernel, version to version, as it gains more features and bulk.

Did FAST-OS succeed? That is a harder question. In the sense of industry impact, we clearly had great success. In the sense of building a vibrant community of OS researchers in DOE, we had some success; there is very good OS work going on at Argonne, Oak Ridge, and Sandia, and on a smaller scale, at other Labs.

Those are hard-won gains, and maintaining them will not be easy: OS research continues to come under pressure for both budget reasons and from those who believe that industry will just supply an OS as a turnkey answer, in spite of all historical evidence to the contrary. No standard OS has ever worked for the large scale without substantial measurement and change.

Further, the challenges in the worlds of Google, Amazon, and Facebook are now much bigger and freer of past constraints: just this year Google announced the Exacycle initiative, and is delivering one billion core hours for researchers. Further, the scale of the commercial world's file storage problems now dwarfs anything that is facing DOE. Hiring and keeping people at DOE Labs is becoming ever more challenging, absent support for new and novel approaches to DOE problems. People are not excited by the idea of spending the next 10 years of their career supporting the legacy software produced over the last quarter century.

DOE needs to build on the success of these programs by continuing to fund new, innovative research that will help vendors set new directions. At the same time, DOE Labs should almost never be in the business of providing production software; nor should commercial uptake of every last bit of research software be the standard by which success is measured. Failure should always be an option. If there are no failures, then the research is not nearly aggressive enough.

When FAST-OS started, the discussion was whether the DOE Labs would have any OS knowledge or capability left by the end of the decade. FAST-OS and its successor program created a culture of competency in the DOE Labs and attracted some very capable commercial and research partners from outside the labs. In the end, we count this research, and the larger program that funded it, as a success. Whether the gains we have made will be maintained is a question that can only be answered at the end of *this* decade.

Bibliography

[1] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. Project kittyhawk: building a global-scale computer: Blue gene/p as a generic computing platform. *SIGOPS Oper. Syst. Rev.*, 42:77–84, January 2008.

[2] Jonathan Appavoo, Amos Waterland, Dilma Da Silva, Volkmar Uhlig, Bryan Rosenburg, Eric Van Hensbergen, Jan Stoess, Robert Wisniewski, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 385–394, New York, NY, USA, 2010. ACM.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. *Translator*, pages 41–46, 2005.

[4] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 189–202. ACM, 1994.

[5] Noah Paul Evans and Eric Van Hensbergen. Brief announcement: Push, a disc shell. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 306–307, New York, NY, USA, 2009. ACM.

[6] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[7] N.C. Hutchinson and L.L. Peterson. The x-kernel: An architecture for implementing network protocols. *Software Engineering, IEEE Transactions on*, 17(1):64–76, 1991.

[8] L. Ionkov and E. Van Hensbergen. Xcpu2: Distributed seemless desktop extension. 2009.

[9] Latchesar Ionkov, Ron Minnich, and Andrey Mirtchovski. The xcpu cluster management framework. In *First International Workshop on Plan9*, 2006.

[10] Venkateswararao Jujjuri, Eric Van Hensbergen, and Anthony Liguori. *VirtFS A virtualization aware File System pass-through.* 2010.

[11] L. Kaplan. Cray cnl. In *FastOS PI Meeting and Workshop*, 2007.

[12] Ronald Minnich and Jim McKie. Experiences porting the plan 9 research operating system to the ibm blue gene supercomputers. *Computer Science - R&D*, 23(3-4):117–124, 2009.

[13] Ronald G. Minnich and John Floren. Using currying and process-private system calls to break the one-microsecond system call barrier. In *Sixth International Workshop on Plan 9*, 2009.

[14] Ronald G. Minnich, John Floren, and Aki Nyrhinen. Measuring kernel throughput on blue gene/p with the plan 9 research operating system. In *Sixth International Workshop on Plan 9*, 2009.

[15] A. Morari, R. Gioiosa, R.W. Wisniewski, F.J. Cazorla, and M. Valero. A quantitative analysis of os noise. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 852–863. IEEE, 2011.

[16] Greg Rodgers, Sandhya Dwarkadas, and Ashwini Nanda. Vertex: A highly scalable software platform for commodity, hybrid, multicore hpc clusters. July 2011.

[17] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[18] Pravin Shinde. Xcpu3: Workload distribution and aggregation. Master's thesis, Vrije Universiteit, Amsterdam, the Netherlands, 2010.

[19] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 371–377, Washington, DC, USA, 2004. IEEE Computer Society.

[20] W.T. Strayer, B.J. Dempsey, and A.C. Weaver. *XTP: The Xpress transfer protocol.* Addison Wesley Longman Publishing Co., Inc., 1992.

[21] Eric Van Hensbergen, Noah Paul Evans, and Phillip Stanley-Marbell. A unified execution model for cloud computing. *SIGOPS Oper. Syst. Rev.*, 44:12–17, April 2010.

[22] Eric Van Hensbergen, Pravin Shinde, and Noah Evans. Poster: Multi-pipes. *OSDI*, 2010.

[23] R.W. Watson. The delta-t transport protocol: Features and experience. In *Local Computer Networks, 1989., Proceedings 14th Conference on*, pages 399–407. IEEE, 1989.

Appendices

NIX

This document will appear in Bell Labs Technical Journal and was the subject of a talk at a Bell Labs Technical Conference in Antwerp, Belgium, Oct. 2011.

Nix: An Operating System For High Performance Manycore Computing

Francisco J. Ballesteros
Noah Evans
Charles Forsyth
Gorka Guardiola
Jim McKie
Ron Minnich
Enrique Soriano

ABSTRACT

This paper describes NIX, an operating system for manycore CPUs. NIX features a heterogeneous CPU model and uses a shared address space. NIX has been influenced by our work on Blue Gene and more traditional clusters. NIX partitions cores by function: Timesharing Cores, or TCs; Application Cores, or ACs; and Kernel Cores, or KCs. One or more TC runs traditional applications. KCs are optional, running kernel functions on demand. ACs are also optional, devoted to running an application with no interrupts; not even clock interrupts. Unlike traditional kernels, functions are not static: the number of TCs, KCs, and ACs can change as needed. Also unlike traditional systems, applications can communicate by sending messages to the TC kernel, instead of system call traps. These messages are "active" taking advantage of the shared-memory nature of manycore CPUs to pass pointers to data and code to coordinate cores.

1. Introduction

Cloud computing uses virtualization on shared hardware to provide each user with the appearance of having a private system of their choosing running on dedicated hardware. Virtualization enables the dynamic provisioning of resources according to demand, and the possibilities of entire system backup, migration, and persistence; for many tasks, the flexibility of this approach, coupled with savings in cost, management, and energy, are compelling.

However, certain types of tasks — such as compute-intensive parallel computations — are not trivial to implement as cloud applications. For instance, HPC tasks consist of highly interrelated subproblems where synchronization and work allocation happen at fixed intervals. Interrupts from the hypervisor and the operating system add “noise” and, thereby, random latency to tasks, slowing down all the other tasks by making them wait for such slowed-down tasks to finish. This slowing effect may cascade and seriously disrupt the regular flow of a computation.

To avoid latency issues, HPC tasks are performed on heavily customized hardware that provides bounded latency. Unlike clouds, HPC systems are typically not time-shared. Instead of the illusion of exclusivity, individual users are given fully private allocations for their task running as a single-user system. However, programming for single user systems is difficult, users prefer programming environments that mimic the time-sharing environment of their desktop. This desire leads to conflicting constraints between the need for a convenient and productive programming environment and the goal of maximum performance. This conflict has led to an evolution of HPC operating systems towards providing the full capabilities of a commodity time-sharing operating system. For example, the IBM Compute Node Kernel, a non-Linux HPC kernel, has changed in the last ten years to become more and more like Linux. On the other hand, Linux systems for HPC are increasingly pared down to the minimal subset of capabilities in order to avoid timesharing degradation at the cost of compatibility with the current Linux source tree. This convergent evolution has lead to an environment where HPC kernels sacrifice performance for compatibility with commodity systems while commodity systems sacrifice compatibility for performance, leaving both issues fundamentally unresolved.

In this paper we describe a solution to this tradeoff, bridging the gap between performance and expressivity, providing bounded latency and maximum computing power on one hand and the rich programming environment of a commodity OS on the other. Based on the reality of coming many-core processors, we provide an environment in which users can be given dedicated, non-preemptable cores on which to run; and in which, at the same time, all the services of a full-fledged operating system are available, this allows us to take the lessons of HPC computing, bounded latency and exclusive use, and apply them to cloud computing. As an example of this approach we present NIX, a prototype operating system for future manycore CPUs. Influenced by our work in High Performance computing, both on Blue Gene and more traditional clusters, NIX features a heterogeneous CPU model and a change from the traditional Unix memory model of separate virtual address spaces. NIX partitions cores by function: Timesharing Cores (TCs); Application Cores (ACs); and Kernel Cores (KCs). There is always at least one TC, and it runs applications in the traditional model. KCs are cores created to run kernel functions on demand. ACs are entirely devoted to running an application, with no interrupts; not even clock interrupts. Unlike traditional HPC Light Weight Kernels, the number of TCs, KCs, and ACs can change with the needs of the application. Unlike traditional operating systems, applications can access OS services by sending a message to the TC kernel, rather than by a system call trap. Control of ACs is managed by means of intercore-calls. NIX takes advantage of the shared-memory nature of manycore CPUs, and passes pointers to both data and code to coordinate among cores.

The paper is organized as follows. First we provide a brief description of NIX and its capabilities. We then evaluate and benchmark NIX under standard HPC workloads. Finally we summarize the work and discuss the applicability of the Nix design to traditional non-HPC applications.

2. NIX

High Performance applications are increasingly providing fast paths to avoid the higher latency of traditional operating system interfaces or eliminating the operating system entirely and running on bare hardware. Systems like Streamline[1] and Exokernels[2] either provide a minimal kernel or a derived fast path that avoids the operating systems functions. This approach is particularly useful for data base systems, fine-tuned servers, and HPC applications.

Systems like the Multikernel [3] and, to some extent, Helios [4] handle different cores (or groups of cores) as different systems, with their own operating system kernels. Sadly, this does not avoid the interference caused by the system on HPC applications.

We took a different path. As machines move towards hundreds of cores on a single chip[5], we believe that it is possible to avoid using the operating system entirely on many cores of the system. In NIX, applications are assigned to cores with no OS interference; that is, without an operating system kernel. In some sense, this is the opposite of the multikernel approach. Instead of using more kernels for more cores, try to use no kernel for (some of) them.

NIX is a new operating system based on this approach, evolving from a traditional operating system in a way that preserves binary compatibility but also enables a sharp break with the past practice of treating manycore systems as traditional SMP systems. NIX is strongly influenced by our experiences over the past five years modifying and optimizing Plan9 to run on HPC systems such as IBM's BlueGene, applying our experience with large scale systems to general purpose computing.

NIX is designed for heterogeneous manycore processors. Discussions with vendors revealed the following trends. First, the vendors would prefer that not all cores run an OS at all times. Second, there is a very real possibility that on some future manycore systems, not all cores will be able to support an OS. This is similar to the approach that IBM is taking with the CellBe[6] and the Wirespeed Processor[7] where primary processors interact with satellite special purposes processors which are optimized for various applications(floating point computation for the CellBe and network packet processing for the Wirespeed Processor)

NIX achieves this objective by assigning specific roles and executables to satellite processors and using a messaging protocol between cores based on shared-memory active messages. These active messages send not just references to data, but also to code. The messaging protocol communicates using a shared memory data structure, containing cache-aligned fields. In particular, these messages contain arguments, a function pointer, to be run by the peer core, and an indication whether or not to flush the TLB (when required).

3. The NIX approach: core roles

There are a few common attributes to current manycore systems. The first is a non-uniform topology, sockets contain CPUs, and CPUs contain cores. Each of these sockets is connected to a local memory, and can reach other memories only via other sockets, via one or more on board networking hops using an interconnection technology like Hypertransport[8] on Quickpath[9]. This leads to a situation where memory access methods are different according to socket locality. The result is a set of non uniform memory access methods, which means that memory access times are also not uniform and unpredictable without knowledge of the underlying interconnection topology.

Although the methods used to access memory are potentially different, the programmatic interface to memory remains unchanged. While the varying topology may affect performance, the structure of the topology does not change the correctness of programs. Such backwards compatibility makes it possible to start from a preexisting code base and to gradually optimize such code to better take advantage of the new structure of manycore systems. Starting from this foundation allows us to concentrate on building applications that solve our particular problems instead of writing an entirely new set of systems software and tools, which would consume much of the effort (like in any system that starts from a clean sheet design). NIX has had a working kernel and a full set of userland code from the start.

By starting with a standard unmodified time-sharing kernel and gradually adding functions to exploit other cores exclusively for either user or kernel intensive processes, means that in the worst case, the time-sharing kernel will behave as a traditional operating system; In the best case, applications will be able to run as fast as permitted by the raw hardware. For our kernel we used Plan 9 from Bell Labs[10] a distributed research operating system amenable to modification.

In addition to a traditional timesharing kernel running on some cores, NIX partitions cores by function, implementing heterogeneous cores instead of a traditional SMP system where every core is equal and running the same copy of the operating system. However, the idea of partitioning cores by function is not novel. The basic x86 architecture has, since the advent of SMP, divided CPUs into two basic types: BSP, or Boot Strap Processor; and AP, or Application Processor[1]. This differentiation means that, in essence, multiprocessor x86 systems have been heterogeneous from the beginning, although this was not visible from the user level to preserve memory compatibility. Many services, like the memory access methods described earlier, maintain the same interface even if the underlying implementation of these methods is fundamentally different. NIX preserves and extends this distinction between cores to handle heterogeneous applications, creating three new classes of cores:

1 The first class, the Timesharing Core (TC) acts as a traditional timesharing system, handling interrupts, system calls, and scheduling. The BSP is always a TC. There can be more than one TC, and a system can consist of nothing but TCs, as determined by the needs of the user. An SMP system can be viewed as a special case, a NIX with only timesharing cores.

2 The second class, the Application Core (AC) runs applications. Only APs can be Application Cores. AC applications are run in a non-preemptive mode, and never field interrupts. On ACs, applications run as if they had no operating system, but they can still make system calls and rely on OS services as provided by other cores. But for performance, running on ACs is transparent to applications, unless they want to explicitly utilize the capabilities of the core.

3 The third class, the Kernel Core (KC), run OS tasks for the TC and are created under the control of the TC. A KC might, for example, run a file system call. KCs never run user mode code. Typical usages for KCs are to service interrupts from device drivers and to perform system calls requested to, otherwise overloaded, TCs.

This separation of cores into specific roles was influenced by discussions with vendors. While cores on current systems are all the homogeneous SMP systems -with a few exceptions, such as the systems developed by IBM mentioned earlier- there is no guarantee of such homogeneity in future systems. Systems with 1024 cores will not need to have all 1024 cores running the kernel, especially given that designers see potential die space and power savings if, for example, a system with *N* cores has only *sqrt*(*N*) cores complex enough to run an operating system and manage interrupts and I/O.

[1] It is actually a bit more complex than that, due to the advent of multicore. On a multicore socket, only one core is the "BSP"; it is really a Boot Strap Core, but the vendors have chosen to maintain the older name.

4. Core Startup

The BSP behaves as a TC and coordinates the activity of other cores in the system. After start-up, ACs sit in a simple command loop, *acsched*, waiting for commands, and executing them as instructed. KCs do the same, but they are never requested to execute user code. TCs execute standard time-sharing kernels, similar to a conventional SMP system.

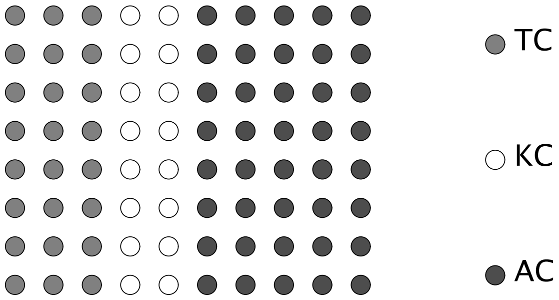


Figure 1 Different cores have different roles. TCs are Time-Sharing cores; KCs are Kernel cores; ACs are Application cores. The BSP (a TC) coordinates the roles for other cores, which may change during time.

Therefore, there are two different types of kernel in the system: TCs and KCs execute the standard time-sharing kernel. ACs execute almost without a kernel, but for a few exception handlers and their main scheduling loop. This has a significant impact on the interference caused by the system to application code, which is negligible on ACs. Nevertheless, ACs may execute arbitrary kernel code by accessing the shared text section of the TC, as all memory is shared in Nix.

Cores can change roles, again under the control of the TC. A core might be needed for applications, in which case NIX can direct the core to enter the AC command loop. Later, the TC might instruct the core to exit the AC loop and re-enter the group of TCs. At present, only one core -the BSP- boots to become a TC; all other cores boot and enter the AC command loop.

5. Inter-Core Communication

In other systems addressing heterogeneous many-core architectures, message passing is the basis for coordination. Some of them handle the different cores as a distributed system[2]. While future manycore systems may have heterogeneous CPUs, one aspect of them it appears will not change: there will still be shared memory addressable from all cores. NIX takes advantage of this property. NIX-specific communications for management are performed via *active* messages, called ‘‘inter-core-calls’’, or ICCs.

An ICC consists of a structure containing a pointer to a function, an indication to flush the TLB, and a set of arguments. Each core has a unique ICC structure associated to it, and polls for a new message while idle. The core, upon receiving the message, calls the supplied function with the arguments given. Note that the arguments, and not just the function, can be pointers, because memory is shared.

The BSP reaches other cores mostly by means of ICCs. Inter-Processor Interrupts (IPIs) are seldom used. They are relegated to cases when asynchronous communication cannot be avoided; for example, when a user wants to interrupt a program running in its AC.

Because of this design, it could be said that ACs do not actually have a kernel. Their ‘‘kernel’’ looks more like a single loop, executing messages as they arrive.

6. User interface

In many cases, user programs may ignore that they are running on NIX and behave as they would in a standard Plan 9 system. The kernel may assign an AC to a process, for example, because it is consuming full scheduling quanta and is considered as CPU bound by the scheduler. In the same way, an AC process that issues frequent system calls might be transparently moved to a TC, or a KC might be assigned to execute its system calls. In all cases, this happens transparently to the process.

For users who wish to maximize the performance of their programs in the NIX environment, manual control is also possible. There are two primary interfaces to the new system functions:

- A new system call:

```
execac(int core, char *path, char *args[]);
```

- Two new flags for the *rfork* system call:

```
rfork(RFCORE); rfork(RFCCORE);
```

Execac is similar to *exec*, but includes a *core* number as its first argument. If this number is 0, the standard *exec* system call is performed, except that all pages are faulted in before the process starts. If this number is negative, the process is moved to an AC, chosen by the kernel. If this number is positive, the process moves to the AC with that core number, if available, otherwise the system call fails.

RFCORE is a new flag for the standard Plan 9 *rfork* system call, which controls resources for the process. This flag asks the kernel to move the process to an AC, chosen by the kernel. A counterpart, *RFCCORE*, may be used to ask the kernel to move the process back to a TC.

Thus, processes can also change their mode. Most processes are started on the TC and, depending on the type of *rfork* or *exec* being performed, can optionally transition to an AC. If the process takes a fault, makes a system call, or has some other problem, it will transition back to the TC for service. Once serviced, the process may resume execution in the AC[2].

Binary compatibility for processes is hence rather easy: old processes only run on a TC, unless the kernel decides otherwise. If a user makes a mistake and starts an old binary on an AC, it will simply move back to a TC at the first system call and stay there until directed to move. If the binary is mostly spending its time on computation, it can still move back to an AC for the duration of the time spent in heavy computation. No checkpointing is needed: this movement is possible because the cores share memory; which means that all cores, whether TC, KC or AC can access any process on any other core as if it was running locally.

[2] Note that this is a description of behavior, and not of the implementation. In the implementation, there is no process migration.

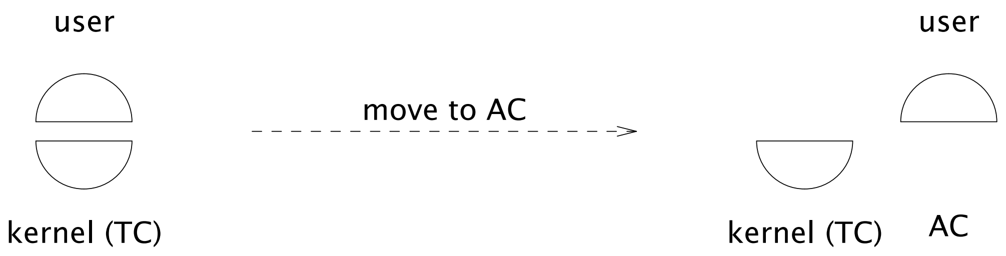


Figure 2 A traditional Plan 9 process has its user space in the same context used for its kernel space. After moving to an AC, the user context is found on a different core, but the kernel part remains in the TC as a handler for system calls and traps.

7. Semaphores and tubes

Because in NIX applications may run in ACs undisturbed by other kernel activities, it is important to be able to perform interprocess communication without needing to resort to kernel calls if feasible. Besides the standard toolkit of IPC mechanisms in Plan 9, NIX includes two mechanisms for this purpose:

- Optimistic user-level semaphores, and
- Tubes, a new user shared-memory communications mechanism.

NIX semaphores use atomic increment and decrement operations, as found on AMD64 and other architectures, to update a semaphore value in order to synchronize. If the operation performed in the semaphore may proceed without blocking (and without needing to wake a peer process from a sleep state), it is performed by the user library without entering the kernel. Otherwise, the kernel is called upon to either block or awake another process. The implementation is simple, with just 124 lines of C in the user library and 310 lines of code in the kernel.

The interface provided for semaphores contains the two standard operations and another one, *altsems*, which is not usual.

```
void upsem(int *sem);
void downsem(int *sem);
int altsems(int *sems[], int nsems);
```

Upsem and *downsem* are traditional semaphore operations, other than their optimism and their ability to run at user-level when feasible. In the worst case, they call two new system calls (*semsleep*, and *semwakeup*) to block or wake up another process, if required. Optionally, before blocking, *downsem* may spin, busy waiting for a chance to perform a down on the semaphore without blocking.

Altsems is a novel operation, which tries to perform a *downsem* in one of the given semaphores (Using *downsem* is not equivalent because it is not known in advance which semaphore will be the target of a down operation). If several semaphores are ready for a down without blocking, one of them is selected and the down is performed; the function returns an index value indicating which one. If none of the downs may proceed, the operation calls the kernel and blocks.

Therefore, in the best case, *altsems* performs a down in user space, without entering the kernel, in a non-determinist way. In the worst case, the kernel is used to *await* for a chance to down one of the semaphores. Before doing so, the operation may be configured to spin and busy wait for a period to wait for its turn.

Optimistic semaphores, as described, are used in NIX to implement shared memory communication channels called *tubes*. A tube is a buffered unidirectional communications channel. Fixed-size messages can be sent and received from it (but different tubes may have different message sizes). The interface is similar to that for Channels in the Plan 9 thread library[10]:

```
Tube* newtube(ulong msz, ulong n);
void freetube(Tube *t);
int nbtrecv(Tube *t, void *p);
int nbtsend(Tube *t, void *p);
void trecv(Tube *t, void *p);
void tsend(Tube *t, void *p);
int talt(Talt a[], int na);
```

Newtube creates a tube for the given message size and number of messages in the tube buffer. *Tsend* and *trecv* can be used to send and receive. There are non-blocking variants, which fail instead of blocking if the operation cannot proceed. And there is a *talt* request to perform alternative sends and/or receives on multiple tubes.

The implementation is a simple producer-consumer written with 141 lines of C, but, because of the semaphores used, it is able to run at user space without entering the kernel when sends and receives may proceed:

```
struct Tube
{
    int msz; /* message size */
    int tsz; /* tube size (# of messages) */
    int nmsg; /* semaphore: # of messages in tube */
    int nhole; /* semaphore: # of free slots in tube */
    int hd;
    int tl;
};
```

It is feasible to try to perform multiple sends and receives on different tubes at the same time, waiting for the chance to execute one of them. This feature exploits *altsems* to operate at user-level if possible, calling the kernel otherwise. It suffices to fill an array of semaphores with either the ones representing messages in a tube, or the ones representing empty slots in a tube, depending on whether a receive or a send operation is selected. Then, calling *altsems* guarantees that, upon return, the operation may proceed.

8. Implementation

As of today, the kernel is operational, although not in production. More work is needed in system interfaces, role changing, and memory management; but the kernel is active enough to be used, at least for testing.

We have changed a surprisingly small amount of code at this point. There are about 400 lines of new assembler source, about 80 lines of platform independent C source, and about 350 lines of AMD64 C source code (not counting the code for NIX semaphores). To this, we have to add a few extra source lines in the start-up code, system call, and trap handlers. This implementation is being both developed and tested on the AMD64 architecture.

As a result of the experiments conducted so far, we found that there seems to be no performance penalty for pre-paging, which is interesting on its own. This is the result of the program image cache, combined with the lack of swapping in NIX. However, not prepaging a binary has performance effects when programs run on ACs, because a TC or KC must be involved in the page fault handling process.

To dispatch a process for execution at one AC we use the ICC mechanism. Figure 3 explains how it works:

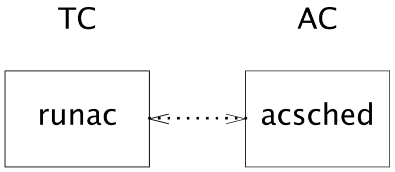


Figure 3 Inter-core calls. The scheduler in the AC is waiting for pointers to functions to be called in the AC context.

The function *acsched* runs on ACs, as part of its start-up sequence. The *acsched* function waits in a fast loop polling for an active message function pointer. To ask the AC to execute a function, the TC sets in the ICC structure for the AC the arguments, an indication to flush the TLB or not, and a function pointer; and then changes the process state to *Exotic*, which would block the process for the moment. When the pointer becomes non-nil, the AC calls the function. The function pointer uses a cache line and all other arguments use a different cache line, in order to minimize the number of bus transactions when polling. Once the function is done, the AC sets the function pointer to nil and calls *ready* on the process that scheduled the function for execution. This approach can be thought of as a software-only IPI.

While an AC is performing an action dictated by a process in the TC, the data structure representing the core, known as its *Mach* structure, points to the process so that the current process pointer, or *up*, in the AC refers to the process. The process refers to the AC via a new field *Mach.ac*. Should the AC become idle, its process pointer in *Mach* is set to nil.

This mechanism is similar to a virtual machine exit sequence. While we could think about implementing AC process execution with a virtual machine startup sequence, virtual machines have significantly more overhead than our method. This mechanism is used by NIX to dispatch processes to ACs, as shown in figure 4:

A process that calls the new system call, *execac* (or uses the *RFCORE* flag in *rfork*) makes the kernel call *runacore* in the context of the process. This function makes the process become a handler for the actual process, which would be running from now on on the AC. To do so, *runacore* calls *runac* to execute *actouser* on the AP selected. This transfers control to user-mode, restoring the state as saved in the user register structure, or *Ureg*, kept by the process in its kernel stack. Both *actouser* and any kernel handler executing in the AC runs using the *Mach* stack, i.e., the per-core stack used in Plan 9 kernels.

The user code runs undisturbed in the AC while the (handler) process is blocked, waiting for the ICC to complete. That happens as soon as there is a fault or a system call in the AC.

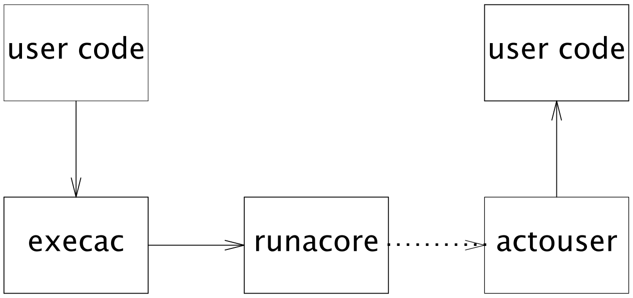


Figure 4 Migration of a user process to an AC using execac.

When an AP takes a fault, the AP transfers control of the process back to the TC, by finishing the ICC, and then waits for directions. That is, the AP spins on the ICC structure waiting for a new call.

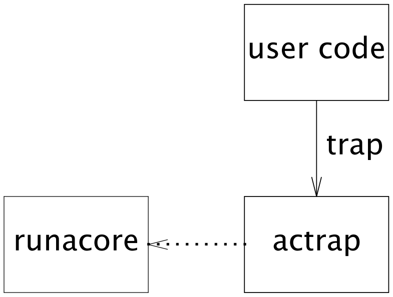


Figure 5 Call path for trap handling in AC context

The handling process, running *runacore*, handles the fault and issues a new ICC to make the AP return from the trap, so that the process continues execution in its core. The trap might kill the process, in which case the AC is released and becomes idle.

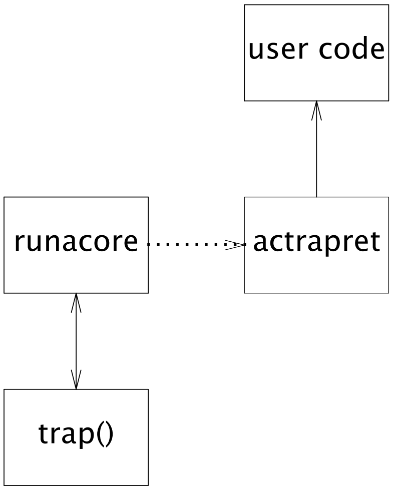


Figure 6 Return path for trap handling in AC context

Handling page faults requires the handling process to get access to the *faulting address*, or *cr2* register in the AMD architecutre, as found in the AC. We have virtualized the register. The TC saves the hardware register into a software copy, kept in the *Mach* structure. The AC does the same. The function *runacore* updates the TC's software cr2 with the one found in the AC before calling *trap*, so that trap handling code does not need to know which core caused the fault.

Floating point traps are handled directly in the AC, instead of dispatching the process to the TC; for efficiency. Only when they cause an event or *note* (the plan 9 equivalent of a unix signal) to be posted to the process, is the process transferred to the TC (usually to be killed).

When an AP makes a system call, the kernel handler in the AP returns control back to the TC in the same manner as page faults, by completing the ICC. The handler process in the TC serves the system call and then transfers control back to the AC, by issuing a new ICC to let the process continue its execution after returning to user mode. As in traps, the user context is kept in the handler process kernel stack, as it is done for all other processes. In particular, it is kept within the *Ureg* data structure as found in that stack.

The handler process, that is, the original time-sharing process when executing *runacore*, behaves like the red line separating the user code (now in the AC) and the kernel code (run in the TC). It is feasible to bring the process back to the TC, as it was before calling *execac*. To do so, *runacore* returns and, only this case, both *execac* and *syscall* are careful not do anything but returning to the caller. The reason is that calling *runacore* is equivalent to returning from *exec* or *rfork* to user code (only that in a different core). All book-keeping to be done while returning, is already done by *runacore*. Also, because the *Ureg* in the bottom of the kernel stack for the process is being used as the place to keep the user context, the code executed after returning from *syscall* restores the user context as it was when the process left the AC to go back to the TC.

Hardware interrupts are all routed to the BSP. ACs should not take any interrupts, as they cause jitter. We changed the round-robin allocation code to find the first core able to take interrupts and route all interrupts to the available core. We currently assume that first core is the BSP. (Note that it is still feasible to route interrupts to other TCs or KCs, and we actually plan to do so in the future). Also, no Advanced Programmable Interrupt Controller(APIC) timer interrupts are enabled on ACs. User code in ACs runs undisturbed, until it faults or makes a system call.

The AC requires a few new assembler routines to transfer control to/from user space, while using the standard *Ureg* space in the bottom of the process kernel stack for saving and restoring process context. The process kernel stack is not used (but for keeping the *Ureg*) while in the ACs; instead, the per-core stack, known as the *Mach* stack, is used for the few kernel routines executed in the AC.

Because the instructions used to enter the kernel, and the sequence, is exactly the same in both the TC and the AC, no change is needed in the C library (other than a new system call for using the new service). All system calls may proceed, transparently for the user, in both kinds of cores.

9. Queue based system calls?

As an experiment, we implemented a small thread library supporting queue-based system calls similar to those in [13]. Threads are cooperatively scheduled within the process and not known by the kernel.

Each process has been provided with two queues: one to record system call requests, and another to record system call replies. When a thread issues a system call, it fills up an slot in the system call queue, instead of making an actual system call. At that point, the thread library marks the thread as blocked and proceeds to execute other threads. When all the threads are blocked, the process waits for replies in the reply queue.

Before using the queue mechanism, the process issues a real system call to let the kernel know. In response to this call, the kernel creates a (kernel) process sharing all segments with the caller. This process is responsible for executing the queued system calls and placing replies for them in the reply queue.

With this implementation, we made several performance measurements. In particular, we measured how long it takes for a program with 50 threads to execute 500 system calls in each thread. For the experiment, the system call used does not block and does nothing. Table 1 shows the result.

Core	System Call	Queue Call
TC	0.02s	0.06s
AC	0.20s	0.04s

Table 1 Times in seconds, of elapsed real time, for a series of syscall calls and queue-based system calls from the TC and the AC.

It takes this program 0.06 seconds (of elapsed, real time) to complete when run on the TC using the queue based mechanism. However, it takes only 0.02 seconds to complete when using the standard system call mechanism. Therefore, at least for this program, the mechanism is more an overhead than a benefit in the TC. It is likely that the total number of system calls per second that could be performed in the machine might increase due to the smaller number of domain crossings. However, for a single program, that does not seem to be the case.

As another experiment, running the same program on the AC takes 0.20 seconds when using the standard system call mechanism, and 0.04 seconds when using queue-based system calls. The AC is not meant to perform system calls. A system call made while running on it implies a trip to the TC and another trip back to the AC. As a result, issuing system calls from the AC is expensive. Looking at the time when using queue-based calls, it is similar to one for running in the TC (but more expensive). Therefore, we may conclude that queue based system calls may make system calls affordable even for ACs.

However, a simpler mechanism is to keep in the TC those processes that did not consume all its quantum at user level, and move to ACs only those processes that do so. As a result, we have decided not to include queue based system calls (although tubes can still be used for IPC).

10. Current Status

There are a few other things that have to be done. To name a few: Including more statistics in the `/proc` interface to reflect the state of the system, considering the different kind of cores in it; deciding if the current interface for the new service is the right one, and to what extent the mechanism has to be its own policy; implementing KCs (which should not require any code, because they must execute the standard kernel); testing and debugging note handling for AC processes; more testing and fine tuning.

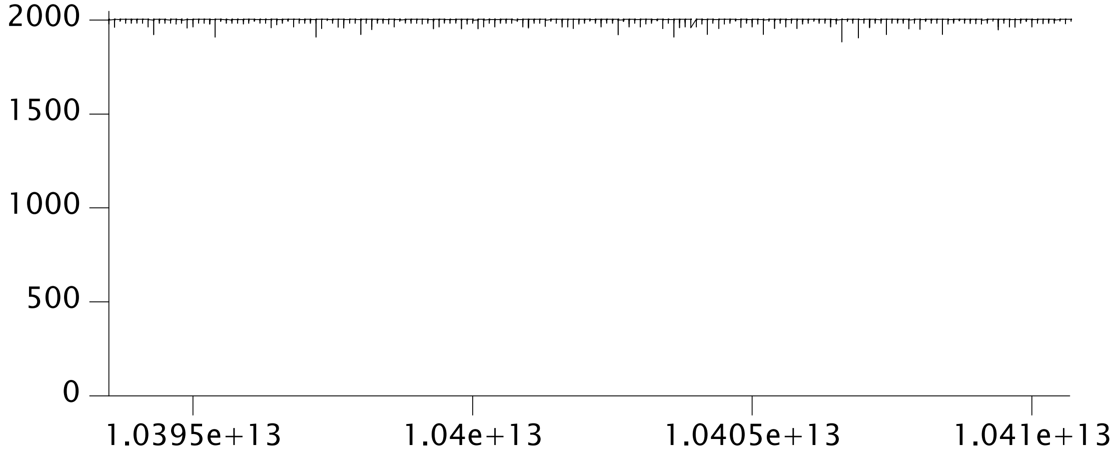
As of today we are implementing new physical memory management and modifying virtual memory management to be able to exploit multiple memory page sizes, as found on modern architectures. Also, a new zero-copy I/O framework is being developed for NIX. We are going to use the features described above, and the ones under construction, to build a high performance file service, along the lines of Venti [14], but capable of exploiting what a many-core machine has to offer, to demonstrate NIX.

11. Evaluation

To show that ACs have inherently less noise than TCs we used the FTQ, or Fixed Time Quantum, benchmark, a test designed to provide quantitative characterization of OS noise[11]. FTQ performs work for a fixed amount of time and then measures how much work was done. Variance in the amount of work done is a result of OS noise.

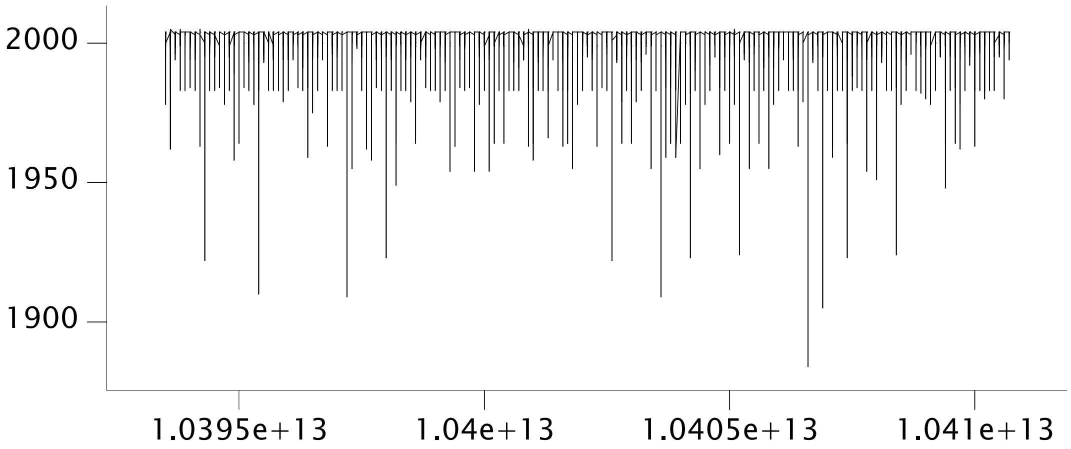
In the following graphs we use FTQ to measure the amount of OS noise on ACs and TCs. Our goal is to maximize the amount of work done while placing special emphasis on avoiding any sort of aperiodic slowdowns in work done. Aperiodicity is a consequence of non-deterministic behaviors in the system that have the potential to introduce a cascading lack of performance as cores slowdown at random. We measure close to 20 billion cycles in 50 thousand cycle increments. The amount of work is measured between zero and just over 2000 arbitrary units.

The first run of the FTQ benchmark on a TC obtains the following results:



FTQ on a TC
Work vs. Cycles
Smoother is better

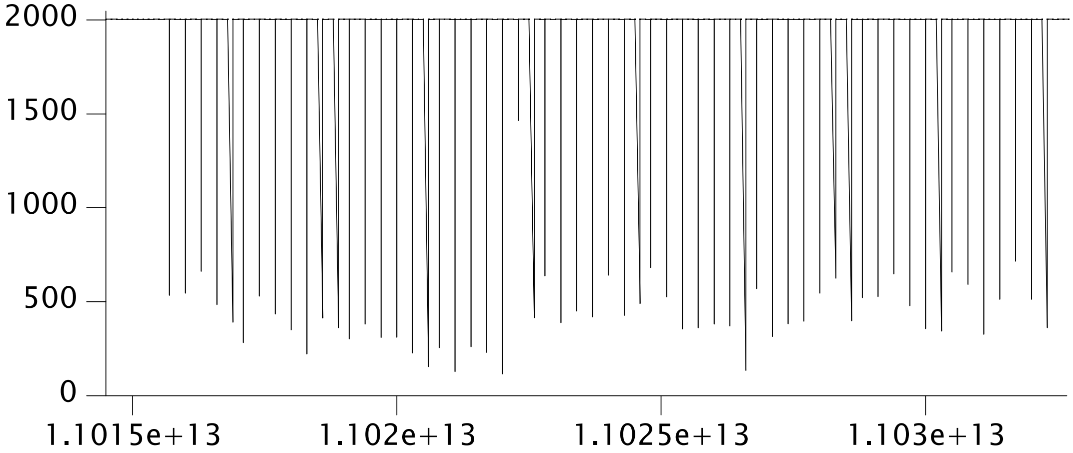
Under closer examination these results reveal underlying aperiodic noise:



FTQ on a TC
Work vs. Cycles
Smoother is better

This noise pattern shows that the TC is unsuitable for any activities which require deterministically bounded latency.

An initial FTQ test on an AC reveals the following:

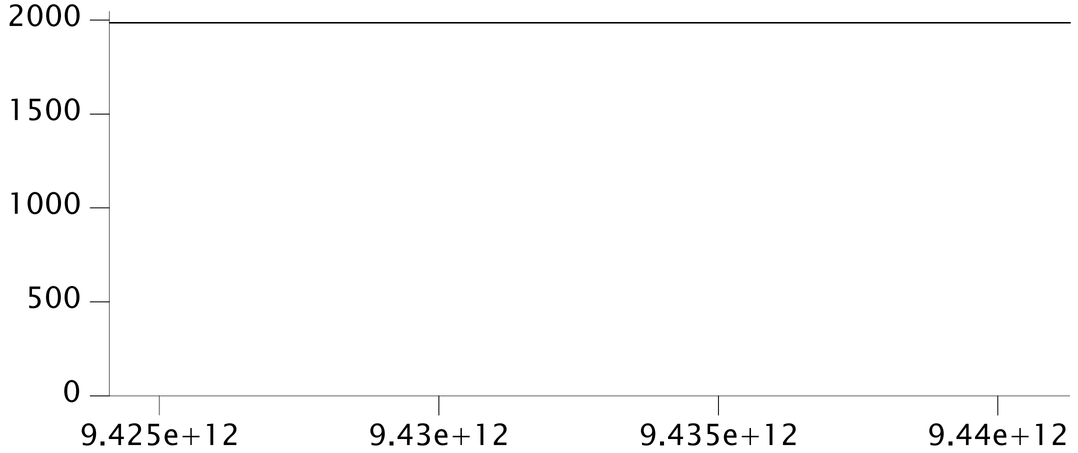


FTQ on an AC
Work vs. Cycles
Smoother is better

The AC is doing the maximum amount of work every quantum but is interrupted by large tasks every 256 quanta. We determined that these quanta coincide with the FTQ task crossing a page boundary. The system is stalling as more data is paged in.

We confirm the origin of this periodic noise by zeroing all of the memory used for FTQ's work buffers. This forces the data pages into memory, eliminating the periodic noise and constantly attaining the theoretical maximum for the FTQ benchmark. This shows the effectiveness of the AC model for computationally intensive HPC tasks.

The final result is shown below:



FTQ on an AC
Work vs. Cycles
Smoother is better

This result is not only very good, it is close to the theoretical ideal. In fact the variance is almost entirely confined to the low-order bit, within the measurement error of the counter.

12. Related Work

The multikernel [3] takes a novel approach for multicore systems. It handles different cores as different systems. Each one is given its own operating system. NIX takes the opposite approach: we try to remove the kernel from the application's core, avoiding unwanted OS interference.

Helios [4] introduces satellite kernels, that run on different cores but still provide the same set of abstraction to user applications. It relies on a traditional message passing scheme, similar to microkernels. NIX also relies on message passing, but it uses active messages and its application cores are more similar to a single loop than they are to a full kernel. Also, NIX does not interfere with applications while in application cores, which Helios does.

NIX does not focus on using architecturally heterogeneous cores like other systems do, for example Barrelfish [15]. In Barrelfish, authors envision using a system knowledge base to perform queries in order to coordinate usage of heterogeneous resources. However, the system does not seem to be implemented and it is not clear that it will be able to execute actual applications in a near future, although it is an interesting approach. NIX took a rather different direction, by leveraging a working system, so that it could run user programs from day zero of its development.

Tessellation [16] partitions the machine, performing both space and time multiplexing. An application is given a partition and may communicate with the rest of the world using messages. In some sense, the tesellation kernel is like an exokernel for partitions, multiplexing partition resources. In contrast, NIX is a full featured operating system kernel. Only that in NIX, some cores may be given to applications. However, applications still have to use standard system services to perform their work.

In a recent Analysis of Linux Scalability to Many Cores [17], authors conclude that "there is no scalability reason to give up on traditional operating system organizations just yet." This supports the idea behind the NIX approach about leveraging a working SMP system and adapting it to better exploit manycore machines. Only that the NIX approach differs from the performance adjustments made to Linux in said analysis.

Fos [18] is a single system image operating system across both multicore and Infrastructure as a Service (IaaS) cloud systems. It is a microkernel system designed to run cloud services. It builds on Xen to run virtual machines for cloud computing. NIX focus is not in IaaS facilities, but on kernel organization for manycore systems.

ROS [19] modifies the process abstraction to become many-core. That is, a process may be scheduled for execution into multiple cores, and the system does not dictate how concurrency is to be handled within the process, which is now a multi-process. In NIX, an application can achieve the same effect by requesting multiple cores (creating one process for each core). Within such process, user-level threading facilities may be used to architect intra-application scheduling. Also, it is not clear if an implementation of ROS ideas exists; as it is not indicated by the paper.

13. Conclusions

NIX presents a new model for operating systems on manycore systems. It follows a heterogeneous multicore model, which is the anticipated model for future systems. It uses active messages for inter-core control. It also preserves backwards compatibility, so that existing programs will always work. Its architecture prevents interference from the OS on applications running on dedicated cores, while at the same time provides a full fledged general-purpose computing system. The approach has been applied to Plan 9 from Bell Labs but, in principle, it can be applied to any other operating system.

14. Acknowledgements

Fabio Pianese provided extensive feedback and clarifications. His help is greatly appreciated.

15. References

[1] Application-tailored I/O with Streamline. W. De Bruijn, H. Bos and H. Bal. ACM Transactions on Computer Systems (TOCS). 2011

[2] Exokernel: An operating system architecture for application-level resource management. D.R. Engler, M.F. Kaashoek and others. ACM SIGOPS Operating Systems Review. 1995

[3] The Multikernel: A new OS architecture for scalable multicore systems. A. Baumann, P. Barham, P.E. Dagand, T Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, A. Singhania. ACM 22nd SOSP. USA. 2009

[4] Helios: Heterogeneous Multiprocessing with Satellite Kernels. E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, G. Hunt. ACM 22nd SOSP. USA. 2009

[5] The single-chip cloud computer. M. Baron. Microprocessor Report. 2010

[6] Cell broadband engine architecture and its first implementation — a performance view. T. Chen, R. Raghavan, J.N. Dale and E. Iwata. IBM Journal of Research and Development. 2007

[7] A wire-speed power™ processor: 2.3 GHz 45nm SOI with 16 cores and 64 threads. C.

Johnson, D.H. Allen, J. Brown and S. Vanderwiel, R. Hoover, H. Achilles, C.Y. Cher, G.A. May, H. Franke, J. Xenedis and others. Solid-State Circuits Conference Digest of Technical Papers (ISSCC). 2010

[8] Hypertransport http://www.hypertransport.org/

[9] Quickpath http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html

[10] Plan 9 From Bell Labs. R. Pike, D. Presotto, K. Thompson, and H. Trickey. Proceedings of the summer 1990 UKUUG Conference 1990

[11] Analysis of microbenchmarks for performance tuning of clusters, M. Sottile, and R. Minnich, Cluster 2004.

[12] The Plan 9 thread library. http://swtch.com/plan9port/man/man3/thread.html

[13] FlexSC: Flexible system call scheduling with exception-less system calls. high Proceedings of the 9th USENIX conference on Operating systems design and implementation. 2010

[14] Venti: a new approach to archival storage. S. Quinlan and S. Dorward. Proceedings of the Conference on File and Storage Technologies. 2002

[15] Embracing diversity in the Barrelfish manycore operating system. Adrian Schupbach, Simon Peter, Andrew Baumann, Timothy Roscoe ACM MMCS 08, USA.

[16] Tessellation: Space-Time Partitioning in a Manycore Client OS Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr†, Krste Asanović, John Kubiatowicz HotPar'09 Proceedings of the First USENIX conference on Hot topics in parallelism.

[17] An Analysis of Linux Scalability to Many Cores Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich OSDI 2010.

[18] An operating system for multicore and clouds: mechanisms and implementation Wentzlaff, David et al 1st ACM symposium on Cloud computing, 2010.

[19] Processes and Resource Management in a Scalable Many-core OS Kevin Klues, Barret Rhoden, Andrew Waterman, David Zhu, Eric Brewer HotPAR 2010.