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.

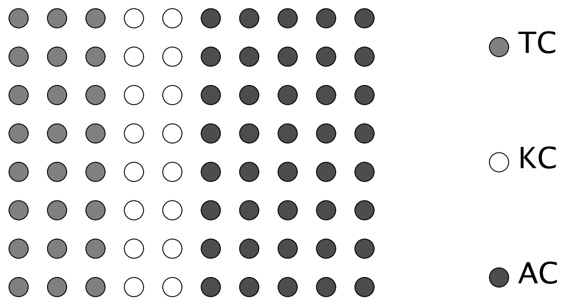


Figure 1 Different cores have different roles. TCs are Time-Sharing cores; KCs are Kernel cores; ACs are Application cores. The BSP (a TC) coordinates the roles for other cores, which may change during time.

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.

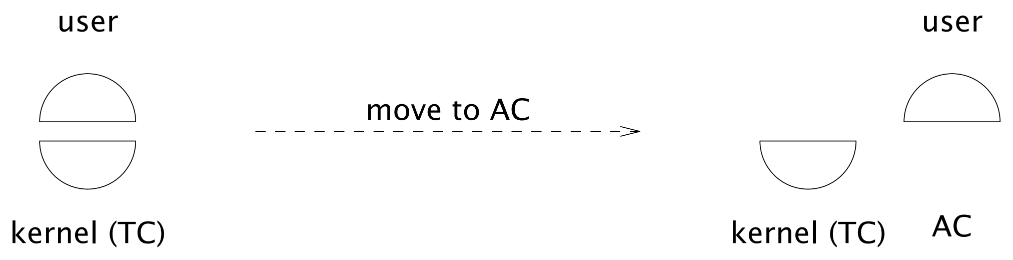


Figure 2 A traditional Plan 9 process has its user space in the same context used for its kernel space. After moving to an AC, the user context is found on a different core, but the kernel part remains in the TC as a handler for system calls and traps.

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:

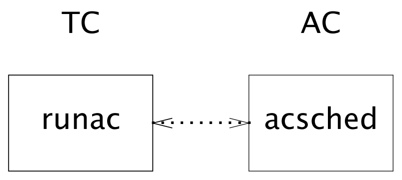


Figure 3 Inter-core calls. The scheduler in the AC is waiting for pointers to functions to be called in the AC context.

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.

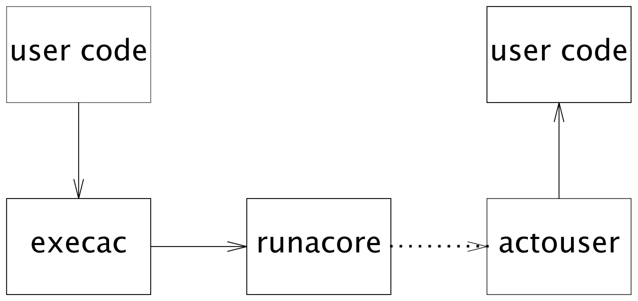


Figure 4 Migration of a user process to an AC using execac.

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.

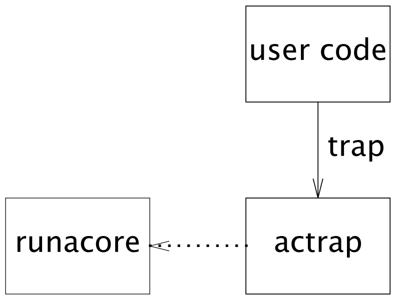


Figure 5 Call path for trap handling in AC context

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.

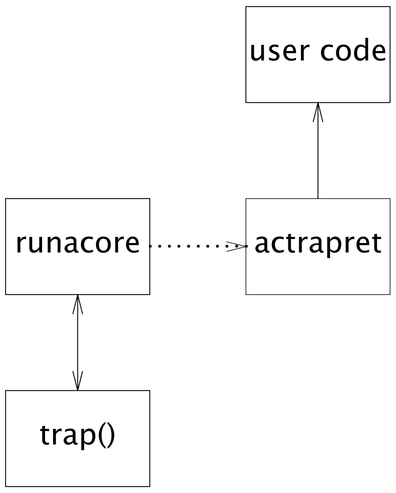


Figure 6 Return path for trap handling in AC context

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.

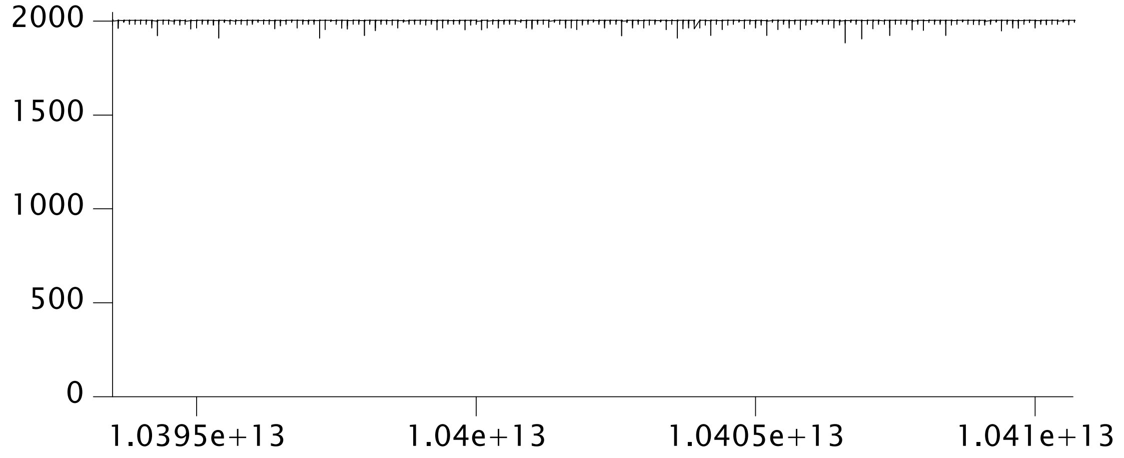
As of today we are implementing new physical memory management and modifying virtual memory management to be able to exploit multiple memory page sizes, as found on modern architectures. Also, a new zero-copy I/O framework is being developed for NIX. We are going to use the features described above, 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.

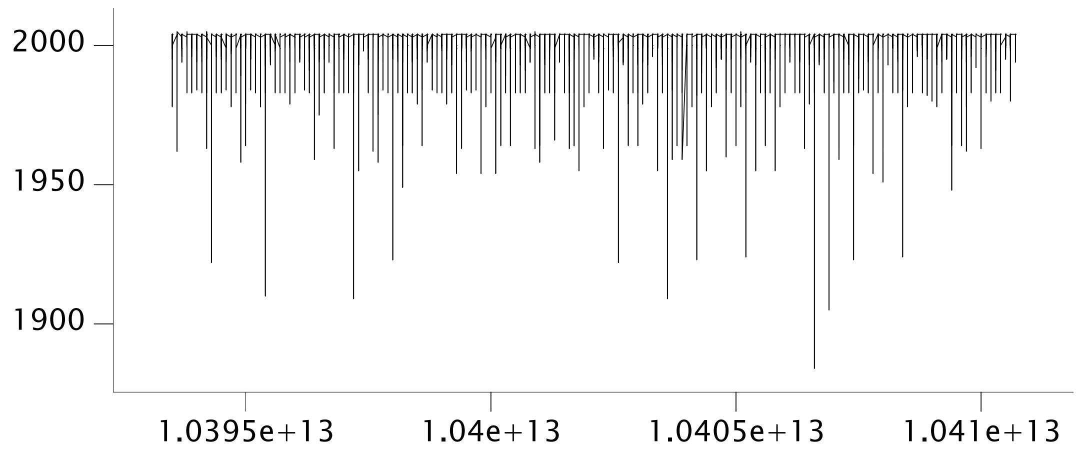
In the following graphs we use FTQ to measure the amount of OS noise on ACs and TCs. Our goal is to maximize the amount of work done while placing special emphasis on avoiding any sort of aperiodic slowdowns in work done. Aperiodicity is a consequence of non-deterministic behaviors in the system that 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

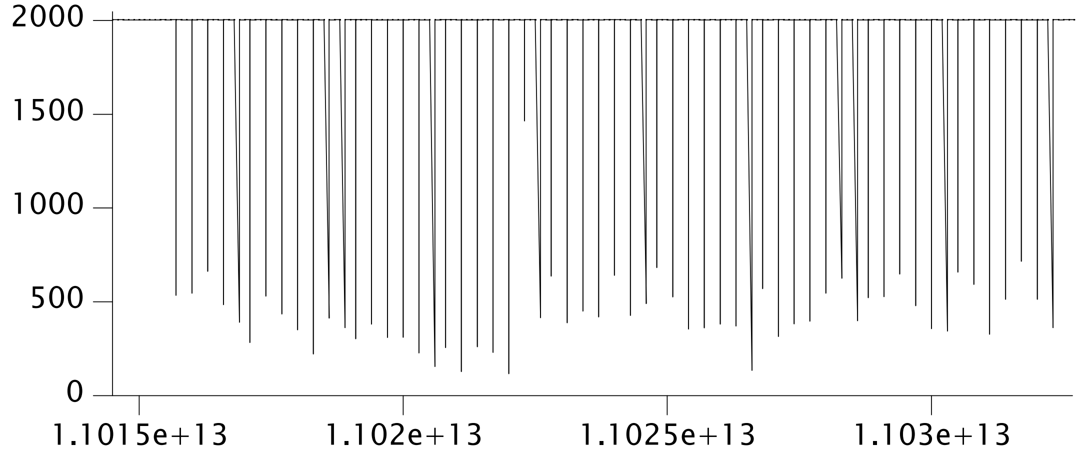
Under closer examination these results reveal underlying aperiodic noise:



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:

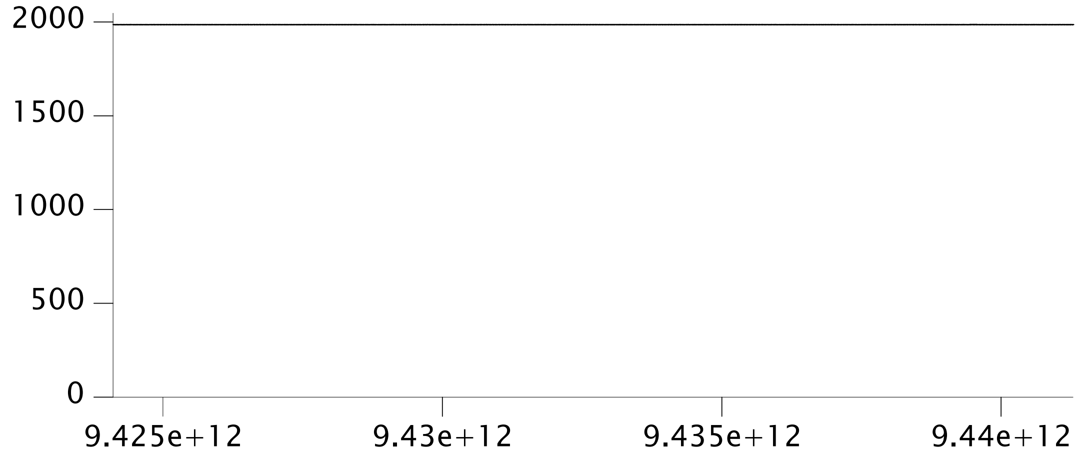


FTQ on an AC
Work vs. Cycles
Smoother is better

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.