NIX, a prototype operating system for manycore CPUs

Ron Minnich
Francisco J. Ballesteros
Gorka Guardiola
Enrique Soriano
Jim McKie
Charles Forsyth
Noah Evans
(want your name here? Just ask!)

ABSTRACT

This paper describes NIX, a prototype operating system for future manycore CPUs. NIX features a heterogeneous CPU model and a change from the traditional Unix memory model of separate virtual address spaces. NIX has been influenced by our work in High Performance computing, both 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. There is always at least one TC, and it runs applications in the traditional model. KCs are optional cores created to run kernel functions on demand. ACs are also optional, and are entirely turned over to running an application, with no interrupts; not even clock interrupts. Unlike traditional HPC Light Weight kernels, functions are not static: the number of TCs, KCs, and ACs can change as needs change. Unlike a traditional operating system, applications can access services by sending a message to the TC kernel, rather than by a system call trap.

Control of ACs is managed by means of active messages. NIX takes advantage of the shared-memory nature of manycore CPUs, and passes pointers to both data and code to coordinate among cores.

1. Introduction

In many cases, it has been argued that the operating system stands in the way of applications and that they cannot increase their performance due to system activities and due to its implementation. In particular, data base systems, fine-tuned servers, applications built using the exokernel approach, with libOSes, all share this view.

However, in the era of multi-core and many-core machines, we believe that we could get rid of the entire operating system kernel. We advocate for a null-kernel or zero-kernel approach, where applications are assigned to cores with no OS interference; that is, without an operating system kernel. Following this idea, we have built NIX.

NIX is a new operating system, evolving from a traditional operating system in a way that preserves binary compatibility but also enables a sharp break with past practice. NIX is strongly influenced by our experiences over the past five years with running Plan 9 on some of the largest supercomputers in the world. It has been said that supercomputing technology frequently tests out ideas that later appear in general purpose computing, and certainly our experience is no exception.

NIX is designed for manycore processors. The manycore is probably obvious at this point. The idea of hetereogeneity derives from discussions we have had with several vendors. 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. We have seen some flavor of this with the Cell which, while a failed effort, does point toward a possible future direction.

NIX uses a messaging protocol between cores based on shared-memory active messages. An active message sends not just references to data, but also to code. The protocol uses a shared memory data structure, containing cache-aligned fields. In particular, it contains arguments, a function pointer, to be run by the peer core, and an indication for flushing TLB (when required).

2. The NIX approach: core roles

There are a few common attributes to todays manycore systems. The systems have a non-uniform topology. Sockets contain CPUs, and CPUs contain cores. Each socket is connected to a local memory, and can reach other memory only via other sockets, via one or more hops; access to memory is hence non-uniform. Core-to-core and core-to-memory communications time is non-uniform.

A very important aspect of these designs is that the varying aspects of the topology affect performance, but not correctness. We preserve an old but very important principle in successful computer projects, both hardware and software: things that worked will still work. We are trying to avoid the high risk of failure that comes with clean sheet designs. NIX thus begins life with a working kernel and full set of userland code.

The idea is to keep a standard time-sharing kernel as it is now, but to be able to exploit other cores for either user or kernel intensive processes. In the worst case, the time-sharing kernel will be able to work as it does now. In the best case, applications will be able to run as fast as permitted by the raw hardware.

NIX derives from Plan 9 from Bell Labs. Source code, compiled for Plan 9, also works on NIX. We make an even stronger guarantee: if a binary worked on Plan 9, it will work on NIX.

NIX partitions cores by function, hence implementing a type of heteregeneity. 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]. In a sense, x86 systems have had heterogeneity from the start, although in keeping with the "things still work" principle, it was not visible for the most part at user level. NIX preserves this distinction, with a twist: we create three new classes of cores:

1 The first, TC, is a time sharing core. The BSP is always a TC. There can be more than one TC, and in fact a system can consist of nothing but TCs, as determined by

[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.

the needs of the user. When all the cores are TC cores, behaviour is not different from a SMP system.

2 The second, AC, is an application core: only APs can be Application Cores. Application cores run applications, in a non-preemptive mode, and never field interrupts. In this case, 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 take control of this feature.

3 The third, KC, is a kernel core. Kernel cores can be created under control of the TC. KCs do nothing but run OS tasks for 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.

Cores hence have roles. Part of the motivation for the creation of these roles comes from discussions with vendors. While cores on current systems are all the same -with a few exceptions, such as Cell- there is no guarantee of such homogeneity in future systems. In fact, heterogeneity is almost guaranteed by a simple fact: systems with 1024 cores will not need to have all 1024 cores running the kernel. Designers see potential for die space and power savings if, say, a system with N cores has only $sqrt(N)$ cores complex enough to run an operating system and manage interrupts and I/O. At least one vendor we have spoken with evinced great dismay at the idea of taking a manycore system and treating it as a traditional SMP system[2].

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.

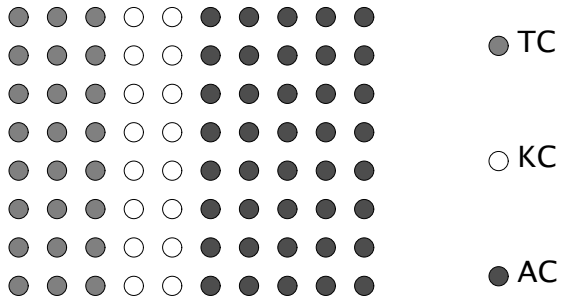


Figure 1 Different cores have different roles. TCs are Time-Sharing cores; KCs are Kernel cores; ACs are Application cores. The BSP (a TC) coordinates the roles for other cores, which may change during time.

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, *acsched*. This has a significant impact on the interference caused by the system to application code, which is

[2] Direct quote: ''It would be a terrible waste to run Linux on every core of this processor, but that's all most people are thinking of''.

negligible on ACs. Nevertheless, ACs may execute arbitrary kernel code as found in the TC, because of the shared memory assumption, if the architecture is compatible.

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. This will be fixed soon.

3. 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.

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 in the current design. 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 we are sharing memory.

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

4. User interface

In many cases, user programs may forget about 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 quanta and is considered as CPU bound. 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 those who care, manual control is also possible. There are two primary interfaces to the new activeity:

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

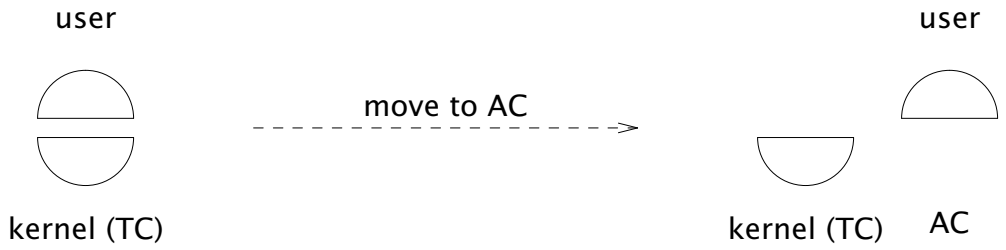


Figure 2 A traditional Plan 9 process has its user space in the same context used for its kernel space. After moving to an AC, the user context is found on a different core, but the kernel part remains in the TC as a handler for system calls and traps.

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 out to an AC for the duration of the time spent in a computational kernel. No checkpointing is needed: this movement works because the cores share memory.

5. Semaphores and tubes

Because in NIX applications may run in ACs undisturbed by other kernel activities, it is important to be able to perform inter process communication wihout the help of the kernel 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.

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 requiring to awake a peer process), it is performed by the user library without entering the kernel. Otherwise, the kernel is called to either block or awake another process.

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* do not deserve any comment, 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. It is not known in advance in which semaphore will be the down operation performed. 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 while.

Optimistic semaphores, as described, are used in NIX to implement shared memory communication channels called *tubes*. A tube is a buffered unidirectional 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:

```
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, 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 at the same time, waiting for the chance to execute one of them. This operation 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.

6. 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. 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 only in the AMD64 architecture.

We found that there seems to be no performance penalty for pre-paging, which is interesting on its own.

To dispatch a process for execution at one AC we use the inter-core-call, active message, mechanism. Figure 3 explains how it works:

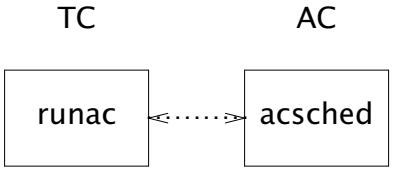


Figure 3 Inter-core calls. The scheduller in the AC is waiting for pointers to functions to be called in the AC context.

The function *acsched* runs on ACs, as part of its start-up sequence. It sits in a tight loop spinning on 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, indication to flush the TLB or not, and the 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, so that polling can be efficient regarding bus transactions. 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. You think of this as a soft IPI.

While an AC is performing an action dictated by a process in the TC, its *Mach* structure points to the process so that *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 *Mach.proc* field is set to nil.

This mechanism is kind of like a vm_exit(). While we could think about implementing this with vm_enter(), it's not quite clear we should take that step. VMs have overhead, at least when compared to the mechanism we use. 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 *Ureg* kept by the

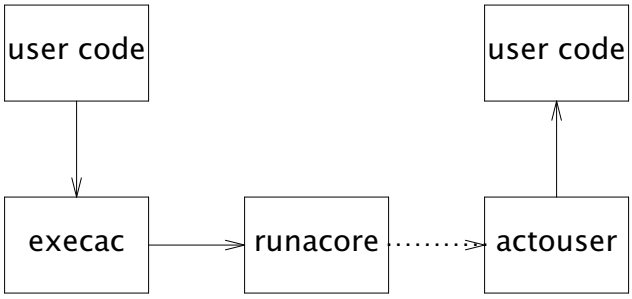


Figure 4 Migration of a user process to an AC using execac.

process in its kernel stack. Both *actouser* and any kernel handler executing in the AC runs using the *Mach* stack.

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.

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.

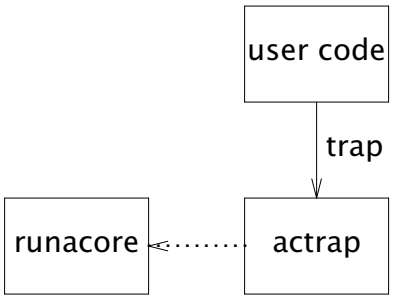


Figure 5 Call path for trap handling in AC context

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.

Handling page faults requires the handling process to get access to the *faulting address*, *cr2* register 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 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* 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 way it is done for 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.

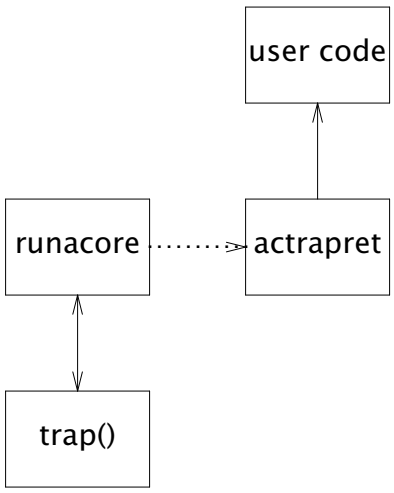


Figure 6 Return path for trap handling in AC context

Like in traps, the user context is kept in the 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* was like returning from *exec* or *rfork* to user code (only that in a different core). All book-keeping to be done while returning, was already done by *runacore*. Also, because the *Ureg* in the bottom of the kernel stack for the process has been used as the place to keep the user context, the code executed after returning from *syscall* would restore the user context as it was when the process left the AC to go back to the TC.

Hardware interrupts are all routed to 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 to that. We assume that first core is the BSP. (Note that is still feasible to route interrupts to other TCs or KCs, and we might do so in the future). Also, no 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 for using the new service). All system calls may proceed, transparently for the user, in both kinds of cores.

7. Queue based system calls?

As an experiment, we implemented a small thread library supporting queue-based system calls similar to those in [5]. Threads are cooperatively scheduled within the process and not known to 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 before proceeding further. 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 more simple 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).

During the experiment, we found an interesting bug in the implementation of NIX. Usually, when a processor has no ready process, the scheduler calls *idlehands* to halt the processor. It will wake up on the next interrupt. However, it can now be the case

than an AC replies to a previous request made by the TC (e.g., to run a user program). By that time, the TC might be idle and it would miss the reply until the next external interrupt. The temporary fix was not to call *idlehands* on the coordinating TC (the BSP).

8. Things not done yet

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.

9. Evaluation

9.1. strid3

Strid3 is part of the "tasty loops" set of benchmarks, and is widely used in HPC evaluation of architectures. In addition to performance measurement it has been used to find bugs in implementations, such as the Barcelona L3 bug.

Strid3 models $S = AX + Y$, or SAXPY, i.e. this loop

```
void saxpy(float* x, float* y, int n, float a) {
  int i;
  for (i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

with a crucial difference: the stride 1 of the above loop is made stride *n*, where n is in the range of 1 to 1024 and is incremented each time through the loop. The loop is also run many times to make sure the computation is long enough to hide startup overheads. The code thus looks like:

```
for( j=0; j<irep; j++ ) {
    t = 1.0/(double)(j+1);
    for( i=0; i<n*incx; i+=incx ) {
        yy[i] += t*xx[i];
    }
}
```

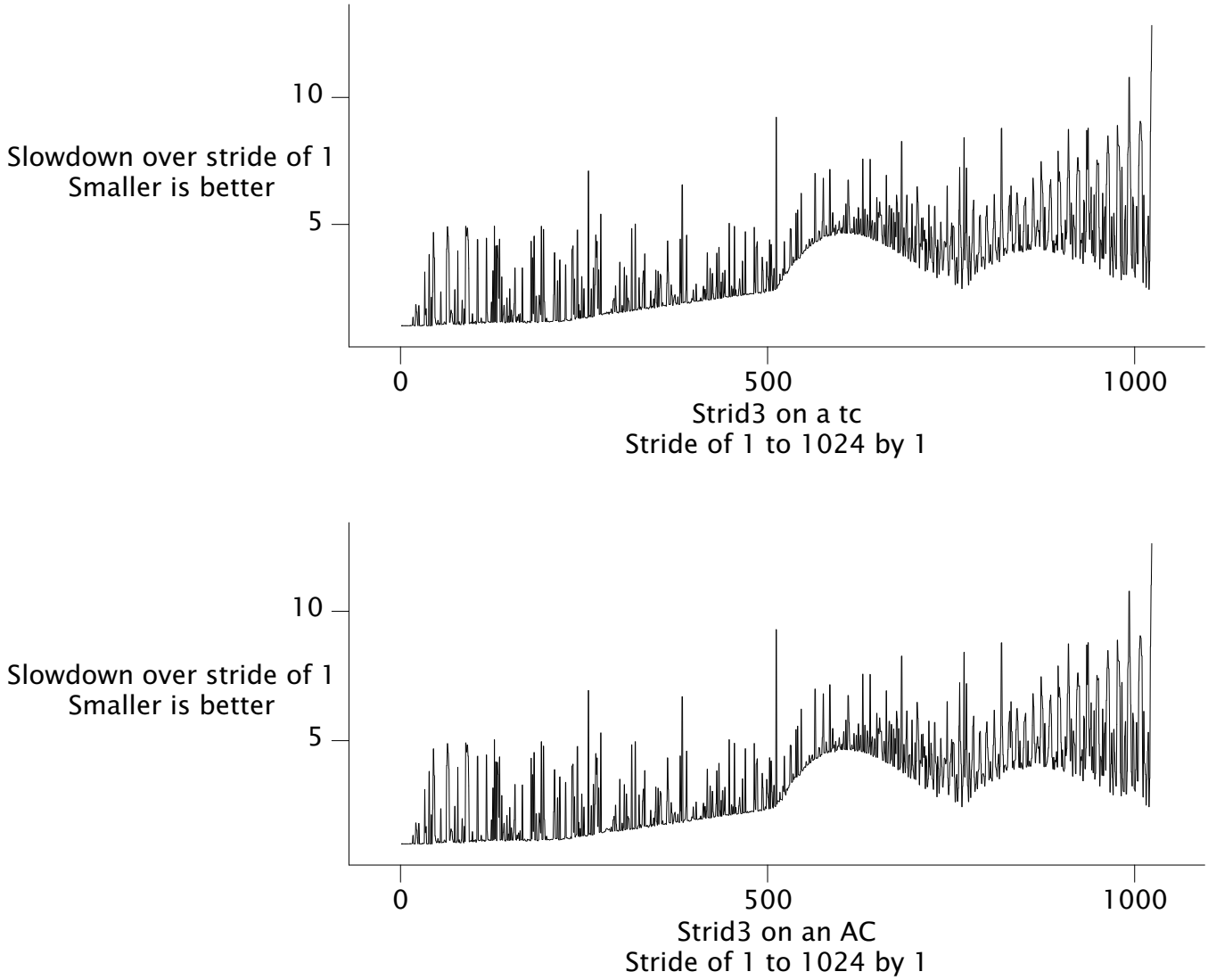
The changing value of *t*, the coeffecient, is done in part to make sure the compiler optimizes nothing away.

In strid3, the innermost loops are shown above. The outer loop is the repeated call to these loops, increasing the stride each time. Each iteration of the outer loop results in a print to stdout.

Strid3 is designed to stress every part of the memory system. For small strides greater than one, one can see impacts as cache lines are used less efficiently. As the stride grows to 256, 512, and beyond, the effect of TLB reloads becomes very visible.

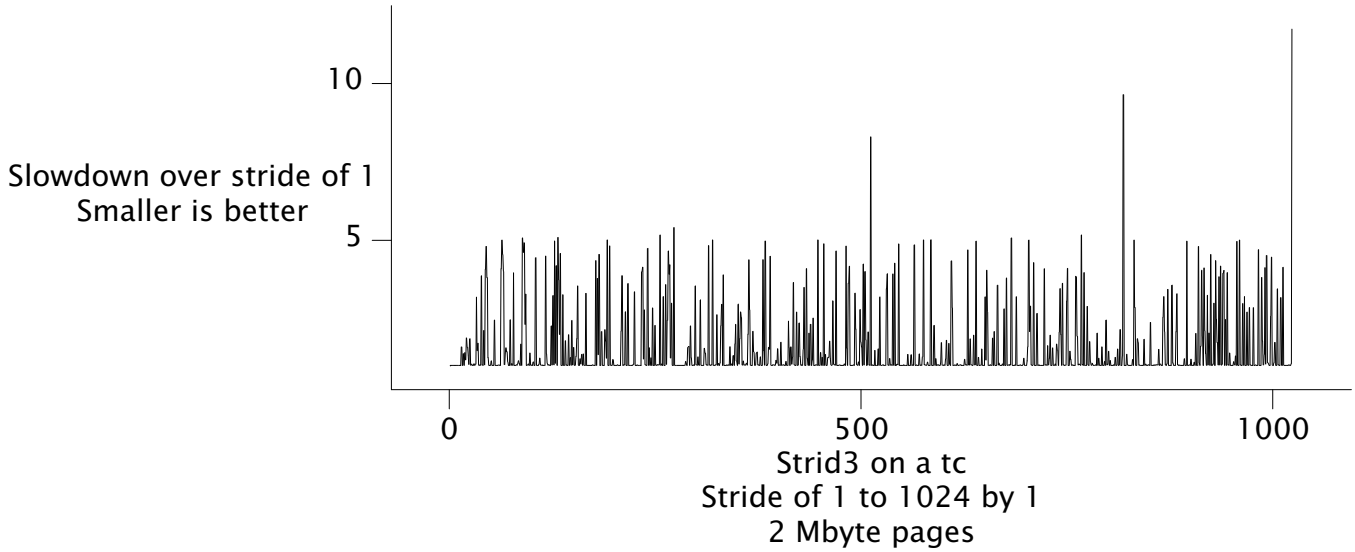
We ran strid3 on the TC and the AC. We had to modfy strid3 in a few simple ways to avoid unnecessary AC to TC transitions that impact performance. The most important change was to avoid the prints in each iteration of the outer loop. The transtion to the TC from the AC was having a small but measurable impact due to the need to take a trap to restart the floating point unit each time the process resumes on the AC.

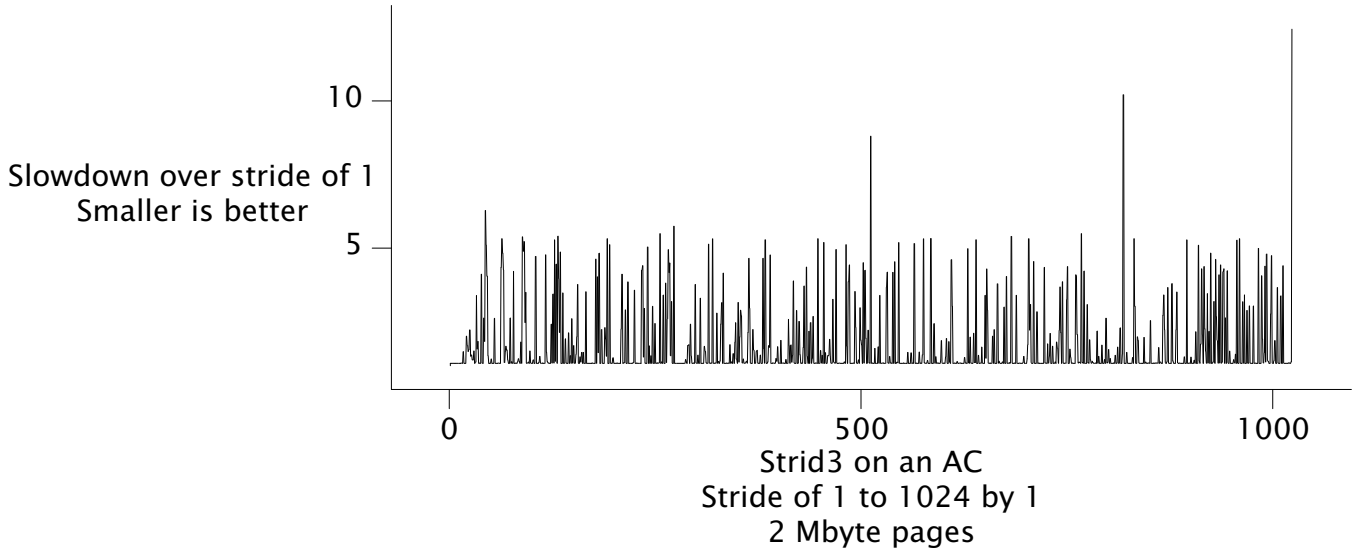
The modified code saves the results in an array and prints them at the end. We show the graph of the results below.



9.2. Two megabyte pages

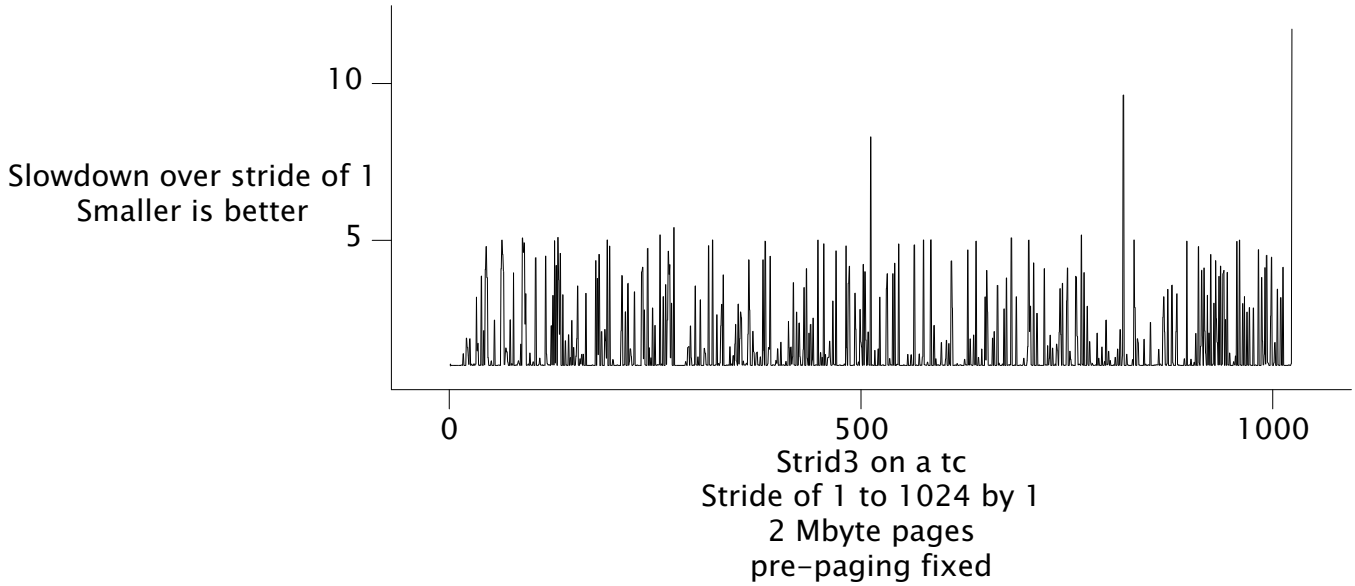
We tested strid3 on a kernel built to support two Mbyte pages. We show the results on tc and ac below.

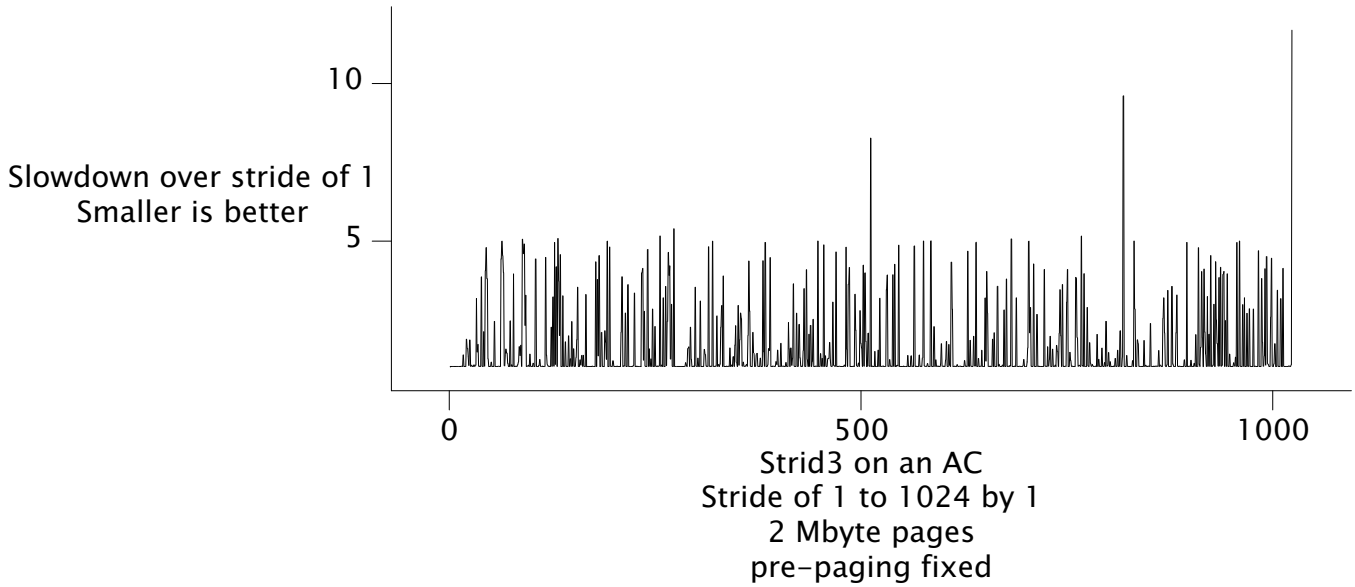




9.2.1. The importance of pre-faulting

As the results show,the time sharing core wins in this case. It turned out there was a bug in the prepaging code. Once the bug was fixed, performance improved as shown below.



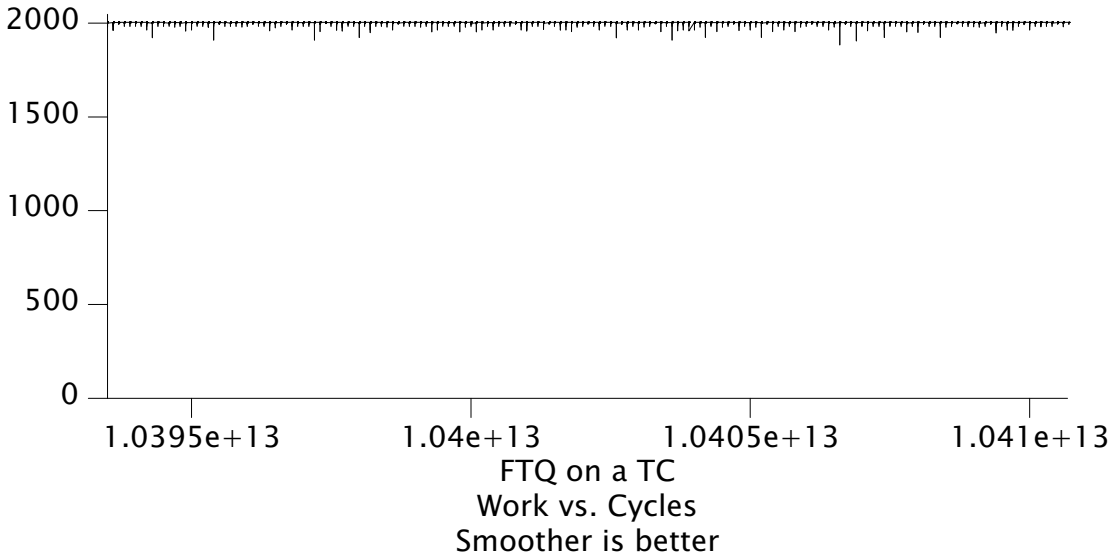


9.3. ftq

FTQ, or Fixed Time Quantum, is a test designed to provide quantitative characterization of OS noise[4]. 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, among other things. Because FTQ measures work per unit of time, and care is taken to make the time measurement stationary, the full set of signal processing tools can be applied to FTQ output data.

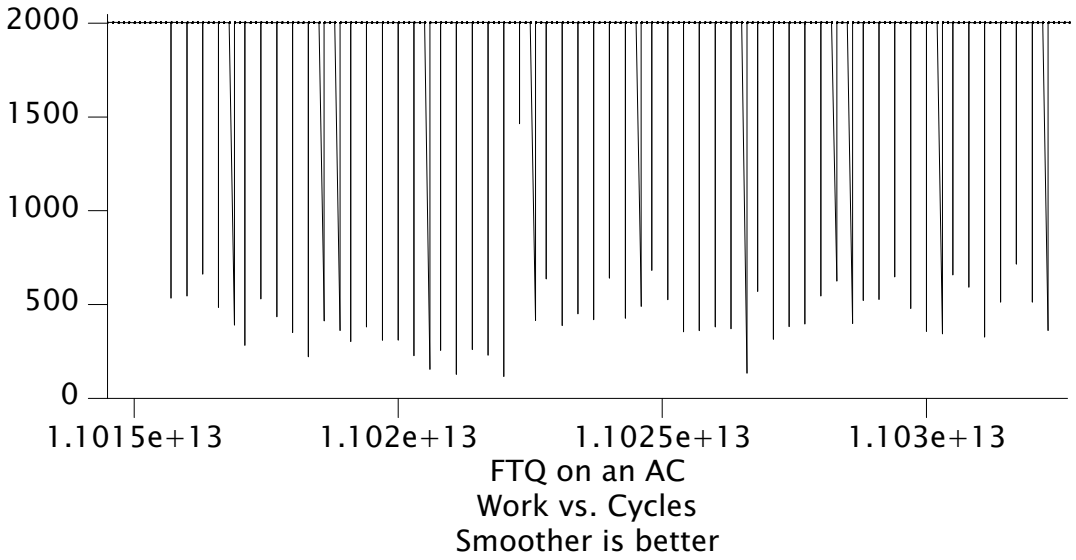
It is possible to eyeball the raw data from FTQ. It is dangerous, as pointed out in the paper, but for simpler examinations it is very useful. A common technique is to plot work performed over time. In an ideal world, the amount of work per time unit is constant. In the real world, it never is. Even on the Blue Gene systems, which are very good, there is always a fair amount of noise due to clock and IO interrupts. We show an FTQ

run on the TC below.



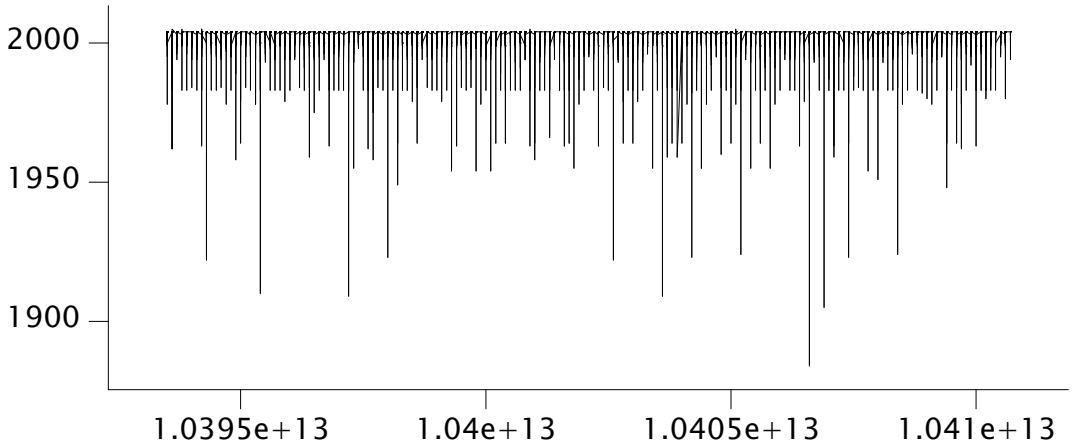
FTQ on a TC
Work vs. Cycles
Smoother is better

Next we show the AC run. It is, apparently, much worse:



FTQ on an AC
Work vs. Cycles
Smoother is better

Here we see the danger of eyeballs: the TC looks better, but in fact is not: the noise is highly aperiodic, which can be seen if we zoom in:



FTQ on a TC
Work vs. Cycles
Smoother is better

Hence, the magnitude of the noise on the AC is much higher, but the noise on the TC has many components. While we might use an FFT or other tools to find the frequency components of the noise, there is a simpler way to quickly diagnose the problem on the AC. Again, observe that the behavior seems strongly periodic. A simple test is to low-pass-filter the data, and then run it through uniq without sorting it. If there is strongly periodic behavior, it will be imediately apparent. The raw data looks like this, showing time absolute stamp (cycle) counter value and work performed:

```
11015693533663 2003
11015694581996 2004
11015695630823 2003
11015696679118 2004
11015715274047 534
11015715553592 2004
11015716602413 2003
11015717650708 2004
11015718699559 2003
11015719747903 2004
```

The low pass filter is trivial: delete the last character of each work performed number, and get rid of the cycle times. The output looks like this:

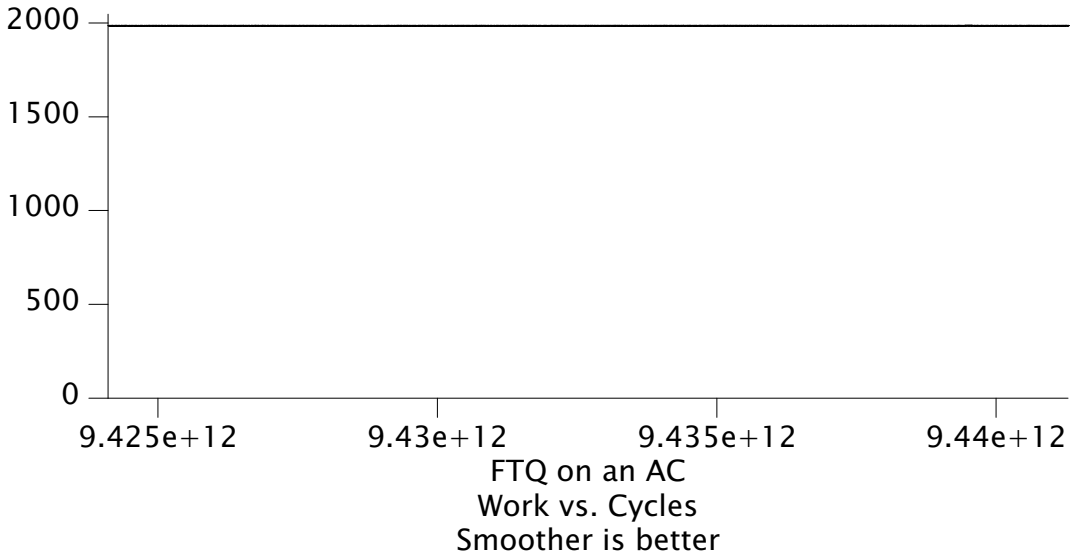
```
200
200
200
200
200
53
200
200
200
200
200
```

When piped through uniq -c, the output looks like this:

```
255 200
  1 54
255 200
  1 66
255 200
  1 48
255 200
  1 39
255 200
  1 28
```

Clearly, something is happening every 256 cycles to slow things down. As it happens, the code uses an array of structs to record the data. Each struct element is 16 bytes. Every 256 elements, the code is moving to a new page. The interference is from page faulting: when we fault, we move back to the TC. In this case, the fault is on an unitialized, allocated array: the fault is to /dev/zero. It is fast but still detectable as interference.

We changed the code to memset the array to zero before moving to the AC. A general rule for using NIX is that programs should mallocz all new data, to make sure pages are in memory, so as to avoid page faults. The result is shown below:



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.

10. Conclusions

NIX presents a new model for operating systems on manycores. 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. Programs will always work.

11. References (for goodness sake let's use refer!)

[1] ISC09 paper
[2] Sc11 paper
[3] KItten paper
[4] FTQ paper
[5] FlexSC paper