

# An analysis of Next Generation Threads on IA64

Ian Wienand

September 9, 2003

## Abstract

For a long time Linux threading support has been solely via the LinuxThreads library. This library is now showing its age and has been often criticised for its lacklustre performance and lack of POSIX compliance. High performance threads are important to the success of the IA64 architecture as many of the CPU and memory intensive applications it is targeted at use threads extensively. Recently, new efforts such as IBM's NGPT and Ulrich Drepper's NPTL have sought to bring high performance POSIX threads to Linux. We compare and contrast the implementation of the old and new libraries and benchmark their performance on Itanium, Pentium and PowerPC based hardware. Our results show that the next generation libraries drastically improve performance of key measures.

This work is supported by UNSW and HP through the Gelato Federation.

<http://www.gelato.org>  
<http://www.gelato.unsw.edu.au>

## 1 Introduction

### 1.1 The IA64 Architecture

The IA64 architecture is the result of a collaboration between HP and Intel to produce a next generation of 64 bit processors. The IA64 architecture combines traditional design principles from RISC, CISC and VLIW designs into a unique package designed to overcome the limitations of these architectures and to scale into the future. Intel's latest incarnation of the IA64 architecture is called the Itanium2.

The Linux port to the IA64 architecture has been actively developed since 1998 [10] and is currently considered to be stable for production use.

### 1.2 Why is threading important?

Since the standardisation of POSIX threads (Pthreads) [8] many applications have been developed to take advantage of the parallelism afforded by threads. For example, the latest release of the Apache web server, Apache 2, uses POSIX threads to increase scalability [1], Java Virtual Machines make extensive use of threads and widely deployed Open Source databases MySQL and PostgreSQL use POSIX threads.

It should be noted Pthreads abstractions are best applied to a uniprocessor or SMP based system; clusters generally use more appropriate libraries such as OpenMP and MPI.

## 2 Overview of Threading Libraries

Multithreading is the ability of an operating system to support multiple threads of execution within a single process [16].

There are a number of ways to organise a threading library (also see Table 1)

1. **Kernel or 1:1** threads refer to a model where the kernel is aware of each thread within a process and participates in its life-cycle (creation, scheduling, removal). This requires support from the kernel as it must understand the relationship between processes and threads. However, it requires minimal library support.

Threading Model	Kernel Support	Library Support	Sample Implementations
Kernel	Extensive	Minimal	LinuxThreads, NPTL
Userspace	None	Extensive	GNU Pth
Hybrid	Some	Some	Solaris ; NGPT

Table 1: A comparison of threading models

2. **Userspace** threads refer to a model where the kernel only sees one process and a userspace library provides the support for threads within that process. This requires no special kernel support but extensive library support.
3. **Hybrid or M:N** threads refer to a model somewhere in between kernel and userspace threads where user space threads may map either to a kernel thread or be managed by the userspace library. This requires both kernel and library support.

Before the Linux kernel fully supported multiple threads there were several userspace libraries available, however, these were generally short lived with the introduction of LinuxThreads.

## 2.1 LinuxThreads

LinuxThreads was originally written by Xavier Leroy and released in 1996, around the time the 2.0 kernel was released. It has remained the dominant Linux thread library for around 8 years and probably has a fair bit of life left in it yet. LinuxThreads unfortunately deviates from the Pthreads standard in some respects and has some serious architectural flaws that fundamentally limit its performance.

The Linux Kernel provides a single interface for creating process and threads; the `clone()` system call. `clone()` was probably first suggested in the Plan9 Operating System [14] and is closely related to the IRIX `sproc()` call. As opposed to providing two unique interfaces for copying a process and for creating a thread, it was realised that `fork()` is simply a special case of thread creation where **more** of the process state is copied. By passing a series of flags to `clone()`, varying levels of process state can be copied (obviously, the major difference when creating a thread is that when a process `fork()`s it receives a new address space, whilst a thread does not).

The `clone()` interface provided by Linux obviously betrays its creator's desired threading model and indeed LinuxThreads is a 1:1 implementation.

LinuxThreads has a number of architectural limitations that hamper its performance [4] [9] :

- **Signals** : LinuxThreads signal infrastructure was initially hampered by a lack of kernel support and consequently deviated from POSIX standards. In brief, POSIX states that any signal sent to a *process* can be handled by any of its threads that does not have the signal blocked. Since using `clone()` makes each thread a unique process as far as the Linux kernel is concerned, if a thread that receives a signal has the signal blocked it will queue for that thread. This also causes problems with signals that are required to stop the entire *process* such as `SIGKILL` or `SIGINT`, which require special kernel support.

LinuxThreads also uses signals to implement some parts of thread synchronisation, which leads to problems with latency and complicates signal handling even further.

- **Limited number of threads** : The number of active threads is maintained in an array of limited size. By default this limits you to 1024 threads. Over the years of development the requirement to search this list was reduced with the implementation of thread registers<sup>1</sup> but needless the limitation remained.

This limit was reasonable on older kernels, as the scheduler would not have dealt with this many threads reasonably. Further, the `/proc` interface was not designed to deal with such high numbers of threads and the fact that the kernel associated a unique PID with each thread meant `/proc` (and associated tools such as `top`) would become almost unusable. The fact that `getpid()` returned a unique PID for each thread also did not correlate with other POSIX implementations.

<sup>1</sup>A thread register is a processor register reserved to point to the current thread. This register is updated by the kernel on context switch and allows a thread to always find out information about its self quickly and easily. This allows thread local storage (TLS) [3], the most useful application of which is the `_thread` attribute for variables which allocates a variable privately for each thread.

- **Manager Thread** : Thread creation and termination require the intervention of a *manager thread*, which does things like allocate stack for the thread and clean up on termination. When spawning many threads this design is an obvious bottleneck.

The manager thread also shows up in debugging sessions and if somehow killed leaves the process in a state that requires manual cleanup. It also causes problems with process accounting, for example the `time` application will not return correct values for multithreaded programs.

LinuxThreads has undergone much development over its life span and provides a reasonable implementation despite its limitations.

## 2.2 Native POSIX Threading Library

The Native POSIX Threading Library (NPTL) is the next generation of POSIX threading for Linux. It has been made possible by significant kernel support developed over the life of the 2.5 development series, and provides significant performance increases across the board. Development was announced in September 2002; the first distribution to include support was Redhat 9 in early April 2003.

Whereas LinuxThreads was forced to work around a lack of kernel support, the clear requirement for high performance POSIX threads had its effect on kernel developers and significant support has been provided in the 2.5 development series. Below we discuss the most important of these changes and how they integrate with NPTL.

### 2.2.1 Futexes

Futexes were introduced by Rusty Russell into the 2.5.7 series kernel and have become an integral part of many applications. Full details about the implementation of futexes can be found in [7].

Just as we can categorise threading models via the level of kernel involvement, we can frame synchronisation primitives the same way. Traditional System V IPC synchronisation techniques such as `semaphores` and `msgqueues` are implemented completely in-kernel and always require a system call when modified.

Pure userspace locking can be provided on an ad-hoc basis with some combination of shared memory, atomic operations and process control but fails as a generic solution. Whilst the actual locking may avoid system calls, as the kernel does not explicitly know about waiting threads it can not make optimal scheduling decisions. Futexes aim to provide the best of both worlds — a standardised interface with the best case not requiring kernel intervention and an efficient waiting mechanism when required. Futexes require user level atomic operations, however these are well supported on most modern architectures.

The actual interface to the futex operations are quite straight forward. The futex system call is prototyped

```
long sys_futex(u32 *uaddr, int op, int val, struct timespec *utime, u32 *uaddr2)
```

`uaddr` is the userspace address that is being used to hold the futex value. `op` and the other arguments vary as below

- **FUTEX\_WAIT** : puts the current processes on the wait queue for this futex. First `val` is check to make sure it is the same as the value in `uaddr` and assuming it is, the process is then queued on the futex. The optional `utime` argument gives a timeout (so timed waits can be implemented).
- **FUTEX\_WAKE** : wakes up `val` number of waiters.
- **FUTEX\_REQUEUE** (since 2.5.70) : will requeue threads waiting on `uaddr` to `uaddr2`. `val` takes the number of processes to wake up, whilst `utime` is overloaded to be the number of waiters to move between the queues. Requeuing allows you to avoid swarming; imagine having two locks `a` and `b`, where there are `n` waiters on `a`. Once `a` is unlocked, all `n` swarm trying to get lock `b`, however only one will get it. The other `n-1` waiters will immediately go onto the wait queue of the second lock.

To fully understand the process, we can look at the locking primitives used to implement mutex's in NPTL. A process locking a mutex through the POSIX standard `pthread_mutex_lock()` interface will end up executing something like Algorithm 1. Note that in the uncontested case there is no need for a system call or even a context switch. In the contested case, we wait on the futex and when woken, test if we have the lock (n.b, the real code obviously makes sure the appropriate parts are atomic).

```

Data      : val, futex
val ← futex;
futex ← futex + 1;
if val equals 0 then
| we have the lock;
else
| repeat
| | val ← val + 1;
| | call sys_futex(futex, FUTEX_WAIT, val, 0, NULL);
| | val ← futex;
| | futex ← futex + 1;
| until val equals 0;
| futex ← 2
end

```

Algorithm 1: Locking a mutex with futexes

The unlock procedure is shown in Algorithm 2. Again the uncontested state does not require any kernel intervention. An interesting design point is that the futex value is always set to zero on unlock. This means that in a situation with  $n$  threads for  $n > 3$ , one with the lock and  $n - 1$  waiting, once the first had given up the lock the others would not be woken. Hence in Algorithm 1 on exit we set  $futex \leftarrow 2$  to ensure we always wake up any threads that are waiting. This reduces the overhead of having to keep track of how many threads are waiting on a futex.

```

Data      : val, futex
val ← futex;
futex ← 0;
if val > 1 then
| call sys_futex(futex, FUTEX_WAKE, 1, 0, NULL);
end

```

Algorithm 2: Unlock a mutex with futexes

The LinuxThreads implementation of the same utilises an adaptive spinlock on the mutex variable, which after a specified number of failures to get the lock uses `sched_yield()` to allow other processes to run.

### 2.2.2 Process and Thread Handling

As mentioned previously, Linux uses the `clone()` call to create both processes *and* threads. The upshot of this is that both threads and processes receive a unique Process ID (PID). Thus the `getpid()` call returns a unique value for threads, which deviates from most other major POSIX threads implementations.

To alleviate this and other problems, a number of new flags have been introduced into the `clone()` system call which are utilised by NPTL.

- **CLONE\_THREAD** : was introduced into the 2.4 series kernels to introduce the concept of *thread groups*. Linux now has a number of different concepts related to processes and threads.
  1. A Process ID **PID** is a unique identifier given to every process and thread.
  2. The Thread Group ID (**TGID**) differentiates threads from processes. When `clone()` is called with `CLONE_THREAD`, the TGID will be set to the *parents* PID. Otherwise, the TGID is set to the PID. Consequently, if the PID is equal to the TGID you have a *process*, if the PID is not equal to the TGID you have a *thread* and the parent process is the PID stored in the TGID. Other PID's in the group are chained in to `pids` (see Figure 1).
  3. Sometimes reference is made to the Thread ID (**TID**) (see the following clone flags for an example). This is the unique value stored in the *userspace* thread descriptor (i.e. the `pthread_t *thread` argument of `pthread_create()`).

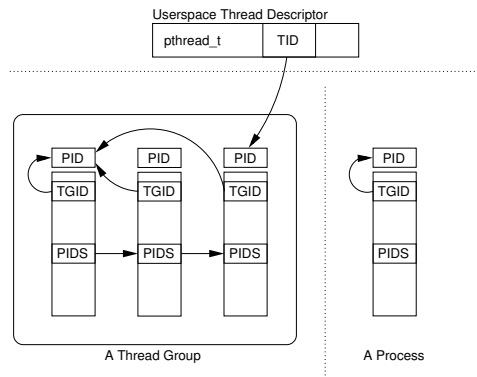


Figure 1: Thread Groups

The `getpid()` call will now return the **TGID**. A new call, `gettid()`, was introduced to return the PID of a thread<sup>2</sup>. Additionally, a new call `exit_group()` was added that cleans up all threads of a process (LinuxThreads leaves this to the manager thread).

Several other extra flags stabilised out of much discussion on the Linux Kernel Mailing List and several revisions. To understand the reason for these flags, consider the `pthread_t` type created with `pthread_create()`. Internally<sup>3</sup> this descriptor has one unique identifier — the Thread ID (**TID**). It was realised that it would be best if the kernel could set the TID value when the thread was cloned. Hence a number of flags were implemented :

- **CLONE\_CHILD\_SETTID** : When this clone flag is set, the kernel will put the PID of the new process to the address passed in the special additional argument `child_tidptr` (i.e. the userspace address of the TID).
- **CLONE\_PARENT\_SETTID** : Like above, but the kernel will put the new PID in the *parent* process. Ulrich Drepper has suggested this would make possible a call like `cfork(int* pid)` (not currently implemented), which returns the PID of the child to the parent processes directly.
- **CLONE\_CHILD\_CLEARTID** : When set, upon `exit()` or `exec()` the address passed in `CLONE_CHILD_SETTID` will be cleared to zero and a `FUTEX_WAKE` will be initiated on the value. This is used by NPTL to implement `pthread_join()`; a thread can `FUTEX_WAIT` on the TID of its child and it will be woken when it completes.

One outstanding (at the time of writing) issue is the `/proc` interface. RedHat 9, which is distributed with NPTL, has integrated patches into its kernels that precedes a threads process ID's with a period, which has the effect of making them hidden files. The standard kernel (as of 2.5.70) shows all threads and processes in `/proc` output. Several patches have been proposed to implement a Solaris style `/proc/pid/tid/` interface, but have so far been rejected for security or scalability reasons.

### 2.2.3 The O(1) Scheduler

The O(1) scheduler was introduced by Ingo Molnar and has been a major part of the 2.5 development series kernels. O(1) refers to the property that no matter how many threads the scheduler has to choose from, it will choose the next one to run in a constant amount of time.

The previous incarnations of the Linux scheduler used the concept of *goodness* to determine which process to run next. All possible runnable tasks are kept on a `runqueue`, which is simply a linked list of processes in a `TASK_RUNNABLE` state. The problem arises that to calculate the next process to run, every possible runnable process must have its goodness calculated and the one with the highest goodness “wins”.

In contrast, the O(1) scheduler uses a `runqueue` structure as shown in Figure 2. The `runqueue` has a number of *buckets* in priority order and a bitmap that flags which buckets have processes available. Finding the next process

<sup>2</sup>Yes, this is confusing! However, it is the path of least resistance as far as the kernel is concerned and a testament to the design of the `clone()` mechanism that it could be implemented so easily.

<sup>3</sup>Applications should never poke around inside this value — POSIX leaves its internal representation completely up to the thread library.

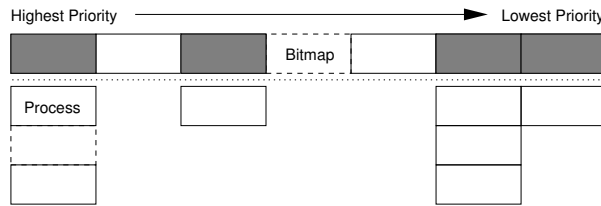


Figure 2: A representation of the runqueue structure in the O(1) scheduler

to run is a matter of reading the bitmap to find the first bucket with processes, then picking the first process off that bucket's queue. The scheduler keeps two such structures, an *active* and *expired* array for processes that are runnable and those which have utilised their entire timeslice respectively. These can be swapped by simply modifying pointers.

The really interesting part, however, is how it is decided where in the runqueue a process should go. The scheduler heuristics have been significantly changed for the O(1) scheduler, resulting in significant performance gains [11]. Some of the advantages include stronger processor affinity (keeping processes tied to the processor they are running on) and better support for interactive programs. Development is ongoing and there appear to be some cases where more work is needed [13].

#### 2.2.4 Signals

*When writing threaded code, treat signals as Jabberwocks – curious and potentially dangerous creatures to be approached with caution, if at all. [2]*

Signal delivery is at best troublesome in a multithreaded environment, and it has been a point of contention between Linux and POSIX. Recent changes have ameliorated these differences and NPTL will be POSIX compliant.

Without the thread group changes above, it was impossible to deliver POSIX compliance in Linux. The POSIX definition of a *process* is a unique entity made up of a number of threads. Linux made every thread a unique entity, so what Linux saw as a *process* POSIX saw as a *thread*. This obviously causes problems anywhere that the rules for a process and a thread are different. For example the `kill(pid_t pid, int sig)` call should send signal `sig` to POSIX process `pid` (and consequently effect all it's threads), but under Linux it ended up sending the signal to the equivalent of a POSIX thread.

Thread groups alleviate this problem. A thread groups maps to the POSIX definition of a *process* and consequently the POSIX rules can be followed. For example, `SIG_STOP` or `SIG_CONT` sent to a process should stop or continue all threads respectively. Without thread groups, only a single thread would receive the signal and the others would not be notified. Similarly, `SIG_KILL` should kill all threads of a process, not just the one it was sent to.

Correct POSIX signal delivery semantics in the kernel are handled via two mechanisms. Correctly shared signal handlers are implemented by the `CLONE_SIGHAND clone()` flag, which makes threads share signal handler structures in the kernel (this flag has been around since 2.0 days). Signal delivery is implemented via a shared signal queue. All process wide signals are placed on a "shared pending" signal queue, from which signals are dequeued and appropriate action can be taken. If the signal will effect the entire process (say, a `SIG_KILL`) all threads in the thread group will be effected. Otherwise a thread is chosen to take the signal, as POSIX defines. This may be a particular thread that is waiting for the signal, or if no thread is waiting any thread that does not have the signal blocked.

There are many corner cases for signal delivery, but recent changes to the 2.5 development series by Ingo Molnar and Roland McGrath have cleaned up signal handling significantly.

### 2.3 Next Generation POSIX Threads

Next Generation POSIX Threads (NGPT) [12] was a POSIX re-implementation of the GNU Pth library [6]. In March 2003 ongoing development was halted, although some support activity remains (see Figure 3).

With the significant kernel changes afforded to NPTL, its performance is outstanding and difficult to beat. Though there has been little extensive testing between the two, initial benchmarks showed that NGPT was fighting

On behalf of the NGPT team, we would like to announce a change in direction for the Next Generation POSIX Threading (NGPT) project.

...

The Linux 2.5 kernel has added many new features in the areas of Scheduler, POSIX signal handling, clone() improvements and futexes that make highly scalable and performing threads a more viable solution in Linux.

...

Our original goal was to make threading in Linux more scalable and POSIX compliant and it seems clear that NPTL has addressed such issues quite well.

Figure 3: An extract from an email explaining the cessation of NGPT development

a losing battle. For example, one would expect that a M:N style implementation should be faster at creating and removing threads, but this was shown to not be the case [5].

The other important factor is not so much technical, but political. No two libraries will be the same and software developers will unfortunately but unquestionably utilise unique features of their chosen library. Causing a fork would only serve to damage general Linux threading portability.

### 3 Benchmarks

We wrote a series of minimal benchmarks to test some common functions of the thread library. Tests consist of taking some action (as described) as many times as possible in a specified time period (5 seconds). For the source code for all benchmarks see <http://www.gelato.unsw.edu.au/IA64wiki/NPTLbenchmarks>. Our goal in testing was firstly to quantify the gains afforded by NPTL and secondly to confirm that IA64 received similar benefits to other architectures.

#### 3.1 Test Framework

All tests were run on an unaltered 2.5.72 kernel, with NPTL release 0.48 (unless otherwise specified). The Linux-Threads minor revisions alter slightly with the machine's distribution, but all come from a 2.3 Glibc.

1. **Intel Pentium 4, 2.5Ghz** : running RedHat 9.
2. **Intel Itanium2 900Mhz** : running Debian unstable.
3. **iBook PowerPC 750FX 700Mhz** : running Debian unstable.

Results are given below in a summarised format; the full results are available for review at the aforementioned web page. Tests were run 10 times for 10 seconds and averaged. Numbers indicate performance of NPTL relative to LinuxThreads, a figure > 1.0 indicates better, < 1.0 would indicate worse performance. Numbers in parentheses indicate the standard deviation in units of the last quoted digit. (E.g. "1.00(1)" means 1.00 with a standard deviation of 0.01, while "1.00(11)" means 1.00 with a standard deviation of 0.11).

#### 3.2 Life Cycle

##### 3.2.1 Description

Test the thread life cycle by creating a thread and joining it as much as we can in a specified time.

##### 3.2.2 Results

Thread Life Cycle	
Platform	Comparison to NPTL
Itanium2	7.74(0)
Pentium4	6.29(1)
PowerPC	7.07(3)

### 3.2.3 Discussion

The thread life cycle is the biggest winner with the new threading libraries. These results stem from the removal of the manager thread and the complex message passing system that was required to setup and destroy a thread under LinuxThreads. The ping pong benchmark in section 3.4 times thread creation but not destruction, so we can infer that the difference between the two refers to the gains from thread destruction.

## 3.3 Context Switch

### 3.3.1 Description

Two threads “fight” over a variable locked by two mutexes. This ends up making the two threads switch as much as they possibly can.

### 3.3.2 Results

Context Switch	
Platform	Comparison to NPTL
Itanium2	2.01(6)
Pentium4	2.89(2)
PowerPC	2.54(3)

### 3.3.3 Discussion

Context switching is up impressively due to a combination of futexes and a smarter scheduler. It is possible these numbers could go even higher for IA64 with newly implemented fast system call paths and other improvements.

## 3.4 Ping Pong

### 3.4.1 Description

“Ping Pong” is a benchmark given in [15] which we modified to run on Linux. The basic parameter to the test is the number of tables at which players will have a virtual rally. Each rally consists of 10,000 hits using mutexes to decide whose turn it is to hit the ball next. The test was run with 500 tables (1000 players or threads).

### 3.4.2 Results

Ping Pong 500 Tables		
Platform	Time to create 1000 threads	Time to complete 500 games
Itanium2	1.98(0)	1.15(5)
Pentium4	8.82(1)	1.18(4)
PowerPC	4.42(6)	1.69(13)

### 3.4.3 Discussion

This test presents a worse case scenario with a thousand threads all constantly locking and context switching. This would be an interesting test of the O(1) scheduler on SMP hardware, as it would need to make many load balancing decisions<sup>4</sup>. In this test the effect of the scheduler is largely removed, so the speed ups are from library enhancements such as futexes and removal of the manager thread. Thread creation time for i386 has increased most spectacularly, this could possibly be attributed to far fewer context switches which are notoriously expensive on i386. IA64 does not show as significant speedup in thread creation time as the other architectures, however comparatively IA64 has a few disadvantages; a much larger register set, register backing store stacks and a default stack size of 32MB (as opposed to 2MB for i386 and 4MB for PowerPC).

<sup>4</sup>Lack of SMP hardware prevented this test and comparison of interactions between NPTL and the older scheduler are not possible in any straightforward manner (as the kernel requirements for NPTL have not been readily backported to 2.4 series kernels).

### 3.5 Uncontested

#### 3.5.1 Description

See how many times a thread can get/release an uncontested lock. This shows the fast path for the locking mechanism.

#### 3.5.2 Results

Uncontested	
Platform	Comparison to NPTL
Itanium2	1.35(0)
Pentium4	1.10(0)
PowerPC	2.38(1)

#### 3.5.3 Discussion

Generally we see smaller performance improvements in this test, as in the uncontested case both NPTL and LinuxThreads should take a userspace fast code path. PowerPC benefits more than others, possibly due to removal of memory barrier constraints or more cache friendliness.

### 3.6 Wakeup

#### 3.6.1 Description

This test runs 10 worker threads with one master thread. The threads conditionally wait on a "queue" that the master thread fills. When the queue is full, do a signal wake, where the woken worker thread processes the queue and re-wakes the master to fill the queue back up. This showing how quickly condition variables respond.

#### 3.6.2 Results

Wake Up	
Platform	Comparison to NPTL
Itanium2	1.21(3)
Pentium4 (NPTL 0.48)	1.26(35)
Pentium4 (Redhat 9)	1.46(10)
PowerPC	1.40(4)

#### 3.6.3 Discussion

We see some interesting effects in this test. The wakeup path of NPTL 0.48 uses the FUTEX\_REQUEUE mechanism discussed above and we see that it gives a small performance hit over a NPTL version that does not use this implementation. This is because another futex call has been added to the unlock path (one to move the queue, another to wake a thread on the moved queue) to handle the worst case scenario.

### 3.7 Effects of Page Size on IA64

#### 3.7.1 Description

IA64 allows a number of different page sizes for the kernel page tables. David Mosberger asked about the difference that page size makes on the above tests. The tests below were run with NPTL 0.36; % Gain refers to how much better NPTL performed over LinuxThreads.

#### 3.7.2 Results

Context Switching				
Page Size	4K	8K	16K	64K
% Gain	54.81	55.16	56.78	52.86

Life Cycle				
Page Size	4K	8K	16K	64K
% Gain	81.82	84.50	87.06	93.97

Wake Up				
Page Size	4K	8K	16K	64K
% Gain	41.61	38.08	39.96	35.31

Uncontested				
Page Size	4K	8K	16K	64K
% Gain	38.01	38.03	38.03	38.03

### 3.7.3 Discussion

We see that larger page sizes are a mixed bag performance wise. Larger pages hurt performance when context switching is involved since you have more state to move about, but give an advantage when doing things like creating a thread which is mostly allocating memory. 16K pages appear to be the “sweet spot” for Itanium; this is indeed the default IA64 Linux page size.

## 4 Conclusion

Although LinuxThreads has been a solid counterpart to most every Linux installation over the last 8 years, its days are limited. NPTL offers greater POSIX compatibility and outperforms LinuxThreads in all our benchmarks, in some cases by over 7 times. IA64 appears to have benefited from the new library on par with other architectures. With the upcoming release of the stable 2.6 series we expect that NPTL will rapidly be deployed as the new standard Linux POSIX threading library.

## References

- [1] Overview of new features in Apache 2.0. [http://httpd.apache.org/docs-2.0/new\\_features\\_2\\_0.html](http://httpd.apache.org/docs-2.0/new_features_2_0.html). Cited 16/03/03.
- [2] David R. Butenhof. *Programming with POSIX Threads*. Addison–Wesley, 1997.
- [3] Ulrich Drepper. ELF handling for Thread-LocalStorage. Available at <http://people.redhat.com/drepper/tls.pdf>. Cited 25/07/03.
- [4] Ulrich Drepper and Ingo Molnar. The native POSIX thread library for Linux. Available at <http://people.redhat.com/drepper/nptl-design.pdf>. Cited 05/06/03.
- [5] Ulrich Drepper and Ingo Molnar. NPTL/NGPT/LinuxThreads thread creation benchmark. <http://people.redhat.com/drepper/perf-s-100000-pro.pdf>. Cited 19/05/03.
- [6] Ralf S. Engelschall. Gnu portable threads. Available at <http://www.gnu.org/software/pth/>. Cited 05/06/03.
- [7] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks : Fast userlevel locking in linux. Ottawa Linux Symposium, 2002.
- [8] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.
- [9] LinuxThreads Frequently Asked Questions. <http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html>. Cited 05/06/03.
- [10] David Mosberger and Stephane Eranian. *IA-64 Linux Kernel*. Prentice Hall PTR, 2002.

- [11] Partha Narayanan. O(1) scheduler benchmarks. From a post to the LKML, see <http://www.kerneltrap.com/node.php?id=343>. Cited 19/05/03.
- [12] Next generation POSIX threading. Available at <http://www-124.ibm.com/pthreads/>. Cited 05/06/03.
- [13] A closer look at the Linux O(1) scheduler. <http://www.hpl.hp.com/research/linux/kernel/o1.php>. Cited 26/05/03.
- [14] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.
- [15] Multithreading in the Solaris operating environment, a technical white paper. <http://www.sun.com/software/whitepapers/solaris9/multithread.pdf>. Cited 25/05/03.
- [16] William Stallings. *Operating Systems*. Prentice Hall International, 3rd edition, 1998.