Linux® Scalability for Large NUMA Systems

Ray Bryant and John Hawkes
Silicon Graphics, Inc.
raybry@sgi.com hawkes@sgi.com

Abstract

The SGI® Altix™ 3000 family of servers and superclusters are nonuniform memory access systems that support up to 64 Intel® Itanium® 2 processors and 512GB of main memory in a single Linux image. Altix is targeted to the high-performance computing (HPC) application domain. While this simplifies certain aspects of Linux scalability to such large processor counts, some unique problems have been overcome to reach the target of near-linear scalability of this system for HPC applications. In this paper we discuss the changes that were made to Linux® 2.4.19 during the porting process and the scalability issues that were encountered. In addition, we discuss our approach to scaling Linux to more than 64 processors and we describe the challenges that remain in that arena.

1 Introduction

Over the past three years, SGI has been working on a series of new high-performance computing systems based on its NUMAflex™ interconnection architecture. However, unlike the SGI® Origin® family of machines, which used a MIPS® processor and ran the SGI® IRIX® operating system, the new series of machines is based on the Intel® Itanium® Processor Family and runs an Itanium version of Linux.

In January 2003, SGI announced this series of machines, now known as the SGI Altix 3000 family of servers and superclusters. As announced, Altix supports up to 64 Intel Itanium 2 processors and 512GB of main memory in a single Linux image. The NUMAflex architecture actually supports up to 512 Itanium 2 processors in a single coherency domain; systems larger than 64 processors comprise multiple single-system image Linux systems (each with 64 processors or less) coupled via NUMAflex into a "supercluster." The resulting system can be programmed using a message-passing model; however, interactions between nodes of the supercluster occur at shared memory access times and latencies.

In this paper, we provide a brief overview of the Altix hardware and discuss the Linux changes that were necessary to support this system and to achieve good (near-linear) scalability for high-performance computing (HPC) applications on this system. We also discuss changes that improved scalability for more general workloads, including changes for high-performance I/O. Plans for submitting these changes to the Linux community for incorporation in standard Linux kernels will be discussed. While single-system-image Linux systems larger than 64 processors are not a configuration shipped by SGI, we have experimented with such systems inside SGI, and we will discuss the kernel changes necessary to port Linux to such large systems. Finally, we present benchmark results demonstrating the results of these changes.

2 The SGI Altix Hardware

An Altix system consists of a configurable number of rack-mounted units, each of which SGI refers to as a brick. Depending on configuration, a system may contain one or more of the following brick types: a compute brick (C-brick), a memory brick (M-brick), a router brick (R-brick), or an I/O brick (P-brick or PX-brick).

The basic building block of the Altix system is the C-brick (see Figure 1). A fully configured C-brick consists of two separate dual-processor systems, each of which is a bus-connected multiprocessor or node. The bus (referred to here as a Front Side Bus or FSB) connects the processors and the SHUB chip. Since HPC applications are often memory-bandwidth bound, SGI chose to package only two processors per FSB in order to keep FSB bandwitdh from being a bottleneck in the system.

The SHUB is a proprietary ASIC that implements the following functions:

- It acts a memory controller for the local memory on the node
- It provides an interface to the interconnection network
- It provides an interface to the I/O subsystem
- It manages the global cache coherency protocol
- It supports global TLB shoot-down and inter-processor interrupts (IPIs)
- It provides a globally synchronized high-resolution clock

The memory modules of the C-brick consist of standard PC2100 or PC2700 DIMMS. With

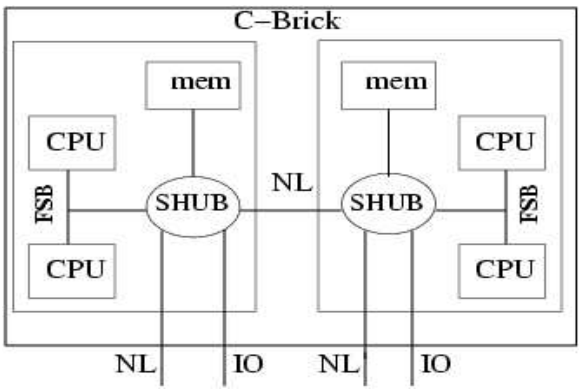

Figure 1: Altix C-Brick

1GB DIMMS, up to 16GB of memory can be installed on a node. For those applications requiring even more memory, an M-brick (a C-brick without processors) can be used.

Memory accesses in an Altix system are either local (i.e., the reference is to memory in the same node as the processor) or remote. Local memory references have lower latency; the Altix system is thus a NUMA (nonuniform memory access) system. The ratio of remote to local memory access times on an Altix system varies from 1.9 to 3.5 depending on the size of the system and the relative locations of the processor and memory module involved in the transfer.

As shown in Figure 1, each SHUB chip provides three external interfaces: two NUMAlink™ interfaces (labeled NL in the figure) to other nodes or the network and an I/O interface to the I/O bricks in the system. The SHUB chip uses the NUMAlink interfaces to send remote memory references to the appropriate node in the system. Depending on configuration the NUMAlink interfaces provide up to 3.2GB/sec of bandwidth in each direction.

I/O bricks implement the I/O interface in the Altix system. (These can be either an IX-brick or a PX-brick. Here we will use the generic term I/O brick.) The bandwidth of the I/O

channel is 1.2GB/sec in each direction. Each I/O brick can contain a number of standard PCI cards; depending on configuration a small number of devices may be installed directly in the I/O brick as well.

Shared memory references to the I/O brick can be generated on any processor in the system. These memory references are forwarded across the network to the appropriate node's SHUB chip and then they are routed to the appropriate I/O brick. This is done in such a way that standard Linux device drivers will work against the PCI cards in an I/O brick, and these device drivers can run on any node in the system.

The cache-coherency policy in the Altix system can be divided into two levels: local and global. The local cache-coherency protocol is defined by the processors on the FSB and is used to maintain cache-coherency between the Itanium processors on the FSB. The global cache-coherency protocol is implemented by the SHUB chip. The global protocol is directory-based and is a refinement of the protocol originally developed for DASH [11] (Some of these refinements are discussed in [10]).

The Altix system interconnection network uses routing bricks (R-bricks) to provide connectivity in system sizes larger than 16 processors. (Smaller systems can be built without routers by directly connecting the NUMAlink channels in a ring configuration.) For example, a 64-processor system is connected as shown in Figure 2.

One measure of the bandwidth capacity of the interconnection network is the bisection bandwidth. This bandwidth is defined as follows: draw an imaginary line through the center of the system. Suppose that each processor on one side of this line is referencing memory on a corresponding node on the other side of this line. The bisection bandwidth is the total amount of data that can be transferred across this line with all processors driven as hard as possible. In the Altix system, the bisection bandwidth of the system is at least 400MB/sec/processor for all system sizes.

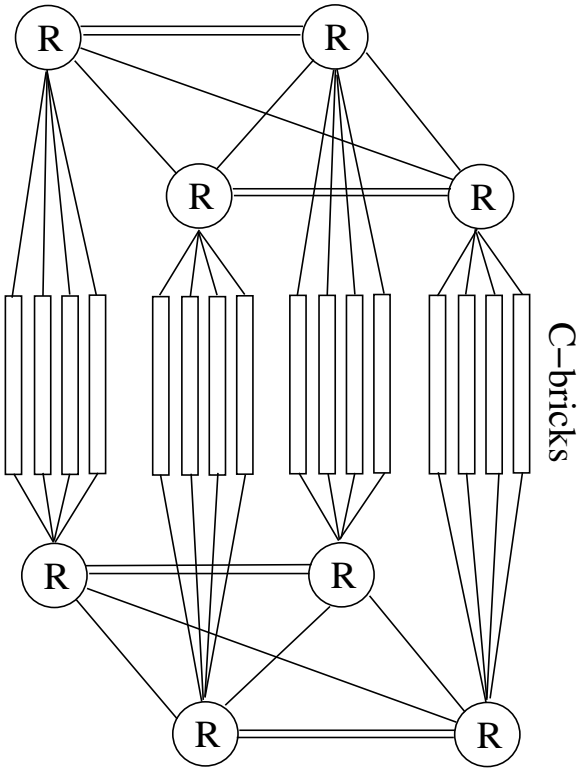

Figure 2: 64-CPU Altix System (R's represent router bricks)

3 Benchmarks

Three kinds of benchmarks have been used in studying Altix performance and scalability:

- HPC benchmarks and applications
- AIM7
- Other open-source benchmarks

HPC benchmarks, such as STREAM [12], SPEC® CPU2000, or SPECrate® [22] are simple, usermode, memory-intensive benchmarks

to verify that the hardware architectural design goals were achieved in terms of memory and I/O bandwidths and latencies. Such benchmarks typically execute one thread per CPU with each thread being autonomous and independent of the other threads so as to avoid interprocess communication bottlenecks. These benchmarks typically do not spend a significant amount of time using kernel services.

Nonetheless, such benchmarks do test the virtual memory subsystem of the Linux kernel. For example, virtual storage for dynamically allocated arrays in Fortran is allocated via *mmap*. When the arrays are touched during initialization, the page fault handler is invoked and zero-filled pages are allocated for the application. Poor scalability in the page-fault handling code will result in a startup bottleneck for these applications.

Once these benchmark tests had been passed, we then ran similar tests for specific HPC applications. These applications were selected based on a mixture of input from SGI marketing and an assessment of how difficult it would be to get the application working in a benchmark environment. Some of these benchmark results are presented in section 7 on page 85.

The AIM Benchmark Suite VII (AIM7) is a benchmark that simulates a more general workload than that of a single HPC application. AIM7 is a C-language program that forks multiple processes (called tasks), each of which concurrently executes similar, randomly ordered set of 53 different kinds of subtests (called jobs). Each subtest exercises a particular facet of system functionality, such as disk-file operations, process creation, user virtual memory operations, pipe I/O, or compute-bound arithmetic loops.

As the number of AIM7 tasks increases, aggregated throughput increases to a peak value. Thereafter, additional tasks produce increasing contention for kernel services, they encounter increasing bottlenecks, and the throughput declines. AIM7 has been an important benchmark to improve Altix scalability under general workloads that consist of a mixture of throughput-oriented runs or of programs that are I/O bound.

Other open-source benchmarks have also been employed. *pgmeter* [6] is a general file system I/O benchmark that we have used to evaluate file system performance[5]. *Kernbench* is a parallel make of the Linux kernel, (e.g., */usr/bin/time make -j 64 vmlinux*). *Hackbench* is a set of communicating threads that stresses the CPU scheduler. Erich Focht's *randupdt* stresses the scheduler's load-balancing algorithms and ability to keep threads executing near their local memory and has been used to help tune the NUMA scheduler for Altix.

4 Measurement and Analysis Tools

A number of measurement tools and benchmarks have been used in our optimization of Linux kernel performance for Altix 3000. Among these are:

- Lockmeter [4, 17], a tool for measuring spinlock contention
- Kernprof [18], a kernel profiling tool that supports hierarchical profiling
- VTune™ [20], a profiling tool available from Intel.
- pfmon [13, 15], a tool written by Stephane Eranian that uses the *perfmon* system calls of the Linux kernel for Itanium to interface with the Itanium processor performance measurement unit

5 Scaling and the Linux Kernel on Altix

"Perfect scaling"—a linear one-to-one relationship between CPU count and throughput for all CPU counts—is rarely achieved because one or more bottlenecks (software or hardware) introduce serial constraints into the otherwise independently parallel CPU execution streams. The common Linux bottlenecks - lock contention and cache-line contention - are true for uniform-memory-access multiprocessor systems as well as for NUMA multiprocessor systems and Altix 3000. However, the performance impact of these bottlenecks on Altix can be exaggerated (potentially in a nonlinear way) by the high processor counts and the directory-based global cache-coherency policy.

Analysis of lock contention has therefore been a key part of improving scalability of the Linux kernel for Altix. We have found and removed a number of different lock-contention bottlenecks whose impacts are significantly worse on Altix than they are on smaller, non-NUMA platforms. Specific examples of these changes are discussed below, in sections 5.2 through 5.8.

Cache-line contention is usually more subtle than lock contention, but cache-line contention can still be a significant impediment to scaling large configurations. These effects can be broadly classified as either "false cache-line sharing" or cache-line "ping-ponging." "False cache-line sharing" is the unintended co-residency of unrelated variables in the same cache-line. Cache-line "ping-ponging" is the change in exclusive ownership of a cache-line as different CPUs write to it.

An example of "false cache-line sharing" is when a single L3 cache-line contains a location that is frequently written and another location that is only being read. In this case, a read will commonly trigger an expensive cache-coherency operation to demote the cache-line from exclusive to shared state in the directory, when all that would otherwise be necessary would be adding the processor to the list of sharing nodes in the cache-line directory entry. Once identified, "false cache-line sharing" can often be remedied by isolating a frequently dirtied variable into its own cache-line.

The performance measurement unit of the Itanium 2 processor includes events that allow one to sample cache misses and record precisely the data and instruction addresses associated with these sampled misses. We have used tools based on these events to find and remove false sharing in user applications, and we plan to repeat such experiments to find and remove false sharing in the Linux kernel.

Some cache-line contention and ping-ponging can be difficult to avoid. For example, a multiple-reader single-writer *rwlock_t* contains an contending variable: the read-lock count. Each *read_lock()* and *read_unlock()* request changes the count and dirties the cache-line containing the lock. Thus, an actively used *rwlock_t* is continually being ping-ponged from CPU to CPU in the system.

5.1 Linux changes for Altix

To get from a `www.kernel.org` Linux kernel to a Linux kernel for Altix, we apply the following open-source community patches:

- The IA-64 patch maintained by David Mossberger [14]
- The "discontiguous memory" patch originally developed as part of the Atlas project [1]
- The O(1) scheduler patch from Erich Focht [7]

- The LSE rollup patch for the Big Kernel Lock [19]

This open-source base is then modified to support the Altix platform-specific addressing model and I/O implementation. This set of patches and the changes specific to Altix comprise the largest set of differences between a standard Linux kernel and a Linux kernel for Altix.

Additional discretionary enhancements improve the user's access to the full power of the hardware architecture. One such enhancement is the CpuMemSets package of library and kernel changes that permit applications to control process placement and memory allocation. Because the Altix system is a NUMA machine, applications can obtain improved performance through use of local memory. Typically, this is achieved by pinning the process to a CPU. Storage allocated by that process (for example, to satisfy page faults) will then be allocated in local memory, using a first-touch storage allocation policy. By making optimal use of local memory, applications with large CPU counts can be made to scale without swamping the communications bandwidth of the Altix interconnection network.

Another enhancement is XSCSI, a new SCSI midlayer that provides higher throughput for the large Altix I/O configurations. Figure 3 shows a comparison of I/O bandwidths achieved at the device-driver level for SCSI and XSCSI on a prototype Altix system. XSCSI was a tactical addition to the Linux kernel for Altix in order to dramatically improve I/O bandwith performance while still meeting product-development deadlines.

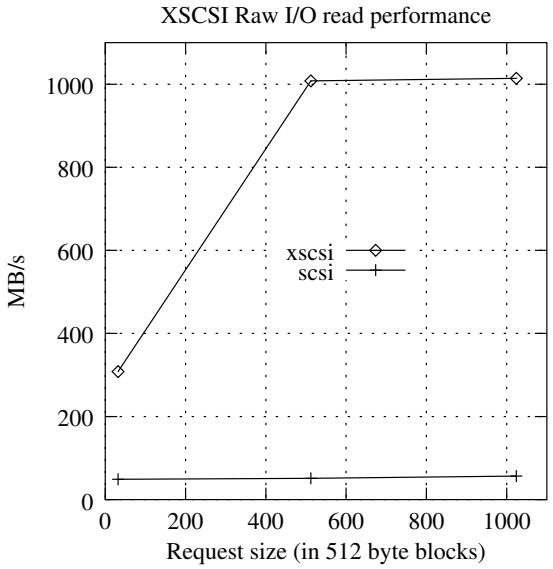

Figure 3: Comparison of SCSI and XSCSI on Prototype Altix Hardware

5.2 Big Kernel Lock

The classic Linux multiprocessor bottleneck has been the *kernel_flag*, commonly known as the Big Kernel Lock (BKL). Early Linux MP implementations used the BKL as the primary synchronization and serialization lock. While this may not have been a significant problem for workloads on a 2-CPU system, a single gross-granularity spinlock is a principal scaling bottleneck for larger CPU counts.

For example, on a prototype Altix platform with 28 CPUs running a relatively unimproved 2.4.17 kernel and with Ext2 file systems, we found that with an AIM7 workload, 30% of the CPU cycles were consumed by waiting on the BKL, and another 25% waiting on the *runqueue_lock*. When we introduced a more efficient multiqueue scheduler that eliminated contention on the *runqueue_lock*, we discovered that the *runqueue_lock* contention simply became increased contention pressure on the BKL. This resulted in 70% of the system CPU cycles being spent waiting on the BKL, up from 30%, and a 30% drop in AIM7 peak throughput performance. The lesson learned here is that we should attack the biggest bottlenecks first, not the lesser bottlenecks.

We have attempted to solve the BKL problem using several techniques. One approach has been in our preferential use of the XFS® file system, vs. Ext2 or Ext3, as the Altix file system. XFS uses scalable, fine-grained locking and largely avoids the use of the BKL altogether. Figure 4 shows a comparison of the relative scalability of several different Linux file systems under the AIM7 workload [5].

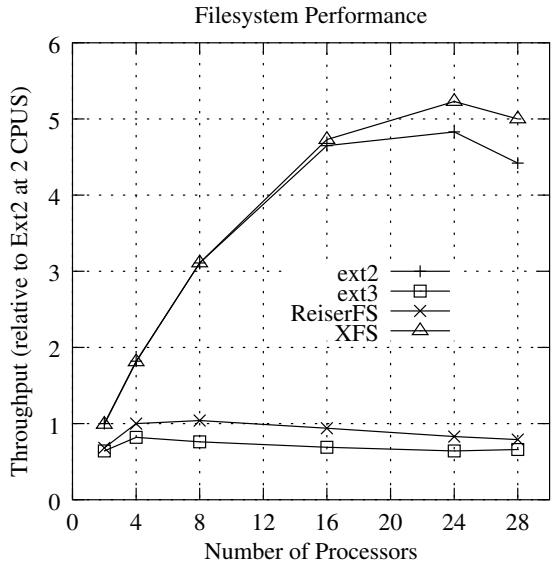

Figure 4: File System Scalability of Linux® 2.4.17 on Prototype Altix Hardware

Other changes were targeted to specific functionality, such as back porting the 2.5 algorithm for process accounting that uses a new spinlock instead of using the BKL.

The largest set of changes was derived from the LSE rollup patch [19]. The aggregate effect of these changes has reduced the fraction of cycles spent spinning (out of all CPU cycles) for the BKL lock from 50% to 5% when running the AIM7 benchmark, with 64 CPUs and XFS filesystems on 64 disks.

5.3 TLB Flush

For the Altix platforms, a global TLB flush first performs a local TLB flush using the Intel *ptc.ga* instruction. Then it writes TLB flush address and size information to special SHUB registers that trigger the remote hardware to perform a local TLB flush of the CPUs in that node. Several changes have been incorporated into the Linux kernel for Altix to reduce the impact of TLB flushing. First, care is taken to minimize the frequency of flushes by eliminating NULL entries in the *mmu_gathers* array and by taking advantage of the larger Itanium 2 page size that allows flushing of larger address ranges. Another reduction was accomplished by having the CPU scheduler remember the node residency history of each thread; this allows the TLB flush routine to flush only those remote nodes that have executed the thread.

5.4 Multiqueue CPU Scheduler

The CPU scheduler in the 2.4 (and earlier) kernels is simple and efficient for uniprocessor and small MP platforms, but it is inefficient for large CPU counts and large thread counts [3]. One scaling bottleneck is due to the use of a single global *runqueue_lock* spinlock to protect the global runqueue, thereby making it heavily contended.

A second inefficiency in the 2.4 scheduler is contention on cache-lines that are involved with managing the global runqueue list. The global list was frequently searched, and process priorities were continually being recomputed. The longer the runqueue list, the more CPU cycles were wasted.

Both of these problems are addressed by new Linux schedulers that partition the single global runqueue of the standard scheduler into multiple runqueues. Beginning with the Linux® 2.4.16 for Altix version, we began to

use the Multiqueue Scheduler contributed to the Linux Scalability Effort (LSE) by Hubertus Franke, Mike Kravetz, and others at IBM [8]. This scheduler was replaced by the O(1) scheduler contributed to the 2.5 kernel by Ingo Molnar [16] and subsequently modified by Erich Focht of NEC and others. The Linux kernel for Altix now uses a version of the 2.5 scheduler with some "NUMA-aware" enhancements and other changes that provide additional performance improvements for typical Altix workloads.

What is relatively peculiar to Altix workloads is the commonplace utilization of the task *cpus_allowed* bitmap to constrain CPU residency. This is typically done to pin processes to specific CPUs or nodes in order to efficiently use local storage on a particular node. The result is that the Altix scheduler commonly encounters processes that cannot be moved to other CPUs for load-balancing purposes. The 2.5 version of the O(1) scheduler's *load_balance()* routine searches only the "busiest" CPU's runqueue to find a candidate process to migrate. If, however, the "busiest" CPU's runqueue is populated with processes that cannot be migrated, the 2.5 version of the O(1) scheduler does not then search other CPUs' runqueues to find additional migration candidates. What is done in the Linux kernel for Altix is that *load_balance()* builds a list of "busier" CPUs and searches all of them (in decreasing order of imbalance) to find candidate processes for migration.

A more subtle scaling problem occurs when multiple CPUs contend inside *load_balance()* on a busy CPU's runqueue lock. This is most often the case when idle CPUs are load-balancing. While it is true that lock contention among idle CPUs is often benign, the problem is that that "busiest" CPU's runqueue lock has now become highly contended, and that stalls any attempt to *try_to_wake_up()* a process on that queue or on the runqueue of any of the idle CPUs that are spinning on another runqueue lock. Therefore, it is beneficial to stagger the idle CPUs' calls to *load_balance()* to minimize this lock contention. We continue to experiment with additional techniques to reduce this contention.

At the time this paper was being written, we have begun to use an adaptation of the O(1) scheduler from Linux® 2.5.68 in the Linux kernel for Altix. To this version of the scheduler, we have added a set of "NUMA-aware" enhancements that were contributed by various developers and that have been aggregated into a roll-up patch by Martin Bligh [2]. Early performance tests show promising results. AIM7 aggregate CPU time is roughly 6% lower at 2500 tasks than without these "NUMA-aware" improvements. We will continue to track further changes in this area and to contribute SGI changes back to the community.

5.5 dcache_lock

In the 2.4.17 and earlier kernels, the *dcache_lock* was a modestly busy spinlock for AIM7-like workloads, typically consuming 3% to 5% of CPU cycles for a 32-CPU configuration. The 2.4.18 kernel, however, began to use this lock in *dnotify_parent()*, and the compounding effect of that additional usage made this lock a major CPU cycle consumer. We have solved this problem in the Linux kernel for Altix by back porting the 2.5 kernel's finer-grained *dparent_lock* strategy. This has returned the contention on the *dcache_lock* to acceptable levels.

5.6 lru_list_lock

One bottleneck in the VM system we have encountered is contention for the *lru_list_lock*. This bottleneck cannot be completely eliminated without a major rewrite of the Linux

2.4.19 VM system. The Linux kernel for Altix contains a minor, albeit measurably effective, optimization for this bottleneck in *fsync_buffers_list()*. Instead of releasing the *lru_list_list* and immediately calling *osync_buffers_list()*, which reacquires it, *fsync_buffers_list()* keeps holding the lock and instead calls a new *__osync_buffers_list()*, which expects the lock to be held on entry. For a highly contended spinlock, it is often better to double the lock's hold-time than to release the lock and have to contend for ownership a second time. This particular change produced 2% to 3% improvement in AIM7 peak throughput.

5.7 xtime_lock

The *xtime_lock* read-write spinlock is a severe scaling bottleneck in the 2.4 kernel. A mere handful of concurrent user programs calling *gettimeofday()* can keep the spinlock's read-count perpetually nonzero, thereby starving the timer interrupt routine's attempts to acquire the lock in write mode and update the timer value. We eliminated this bottleneck by incorporating an open-source patch that converts the *xtime_lock* to a lockless-read using *frlock_t* functionality (which is equivalent to *seqlock_t* in the 2.5 kernel).

5.8 Node-Local Data Structures

We have reduced memory latencies to various per-CPU and per-node data structures by allocating them in node-local storage and by employing strided allocation to improve cache efficiency. Structures where this technique is used include *struct cpuinfo_ia64*, *mmu_gathers*, and *struct runqueue*.

6 Beyond 64 Processors

The Altix platform hardware architecture supports a cache-coherent domain of 512 CPUs. Although SGI currently supports a maximum of 64 processors in a single Linux image, we believe there is potential interest in SSI Linux systems that are larger than 64 CPUs (judging from our experience with IRIX systems, where some customers run 512-CPU SSI systems). Additionally, testing on systems that are larger than 64 CPUs can help us find scalability problems that are present, but not yet obvious in smaller systems.

The Linux kernel currently defines a CPU bit mask as an *unsigned long*, which for an Itanium architecture provides enough bits to specify 64 CPUs. To support a CPU count that exceeds the bit count of an *unsigned long* requires that we define a *cpumask_t* type, declared as *unsigned long[N]*, where *N* is large enough to provide sufficient bits to denote each CPU. While this is a simple kernel coding change, the change affects numerous files. Moreover, it also affects some calling sequences which today expect to pass a *cpumask_t* as a call-by-value argument or as a simple function return value. Most problematic is when *cpumask_t* is involved in a user-kernel interface, such as we have with the SGI CpuMemSets functions like *runon* and *dplace*. Our plan is to follow the approach of the 2.5 kernel in this area (c.f., *sched_set/get_affinity*).

Our initial 128-CPU investigations have so far not yielded any great surprises. The Altix hardware architecture scales memory access bandwidths to these larger configurations, and the longer memory access latencies are small (65 nanoseconds per router "hop" in the current systems) and well understood.

The large NR_CPU configurations benefit from anti-aliasing the per-node data and from dynamically allocating the *struct cpuinfo_ia64* and *mmu_gathers*. Dynamic allocation of the *runqueue_t* elements reduces the static size of the kernel, which otherwise produces a "gp

overflow" at link time. Inter-processor interrupts and remote SHUB references are made in physical addressing mode, thereby avoiding the use of TLB entries and reducing TLB pressure. Finally, several calls to *__cli()* were eliminated or converted into locks.

7 HPC Application Benchmark Results

In this section, we present some benchmark results for example applications in the HPC market segment.

7.1 STREAM

The STREAM Benchmark [12] is a "simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/sec) and the corresponding computation rate for simple vector kernels" [12]. Since many HPC applications are memory bound, higher numbers for this benchmark indicate potentially higher performance on HPC applications in general. The STREAM Benchmark has the following characteristics:

- It consists of simple loops
- It is embarrassingly parallel
- It is easy for compiler to generate scalable code for this benchmark
- In general, only simple optimization levels are allowed

Here we report on what is called the "triad" portion of the benchmark. The parallel Fortran code for this kernel is shown below:

```
!$OMP PARALLEL DO
    DO j = 1, n
       a(j) = b(j) + s*c(j)
    CONTINUE
```

To execute this code in parallel on an Altix system, the data is evenly divided among the nodes, and each processor is assigned to do the calculations on the portion of the data on its node. Threads are pinned to processors so that no thread migration occurs during the benchmark. The scalability results for this benchmark on Altix 3000 are as shown in figure 5. As can be seen from this graph,

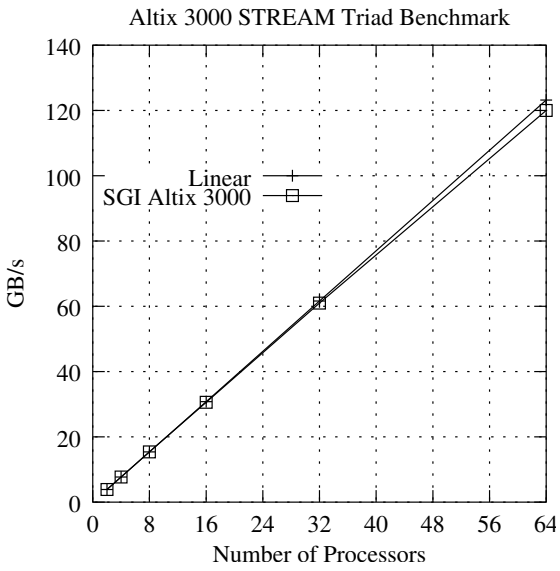

Figure 5: Scalability of STREAM TRIAD benchmark on Altix 3000

the results scale almost linearly with processor count. In some sense, this is not surprising, given the NUMA architecture of the Altix system, since each processor is accessing data in local memory. One can argue that any non-shared memory cluster with a comparable processor could achieve a similar result. However, what is important to realize here is that not all multiprocessor architectures can achieve such a result. A typical uniform-memory-access multiprocessor system, for example, could not achieve such linear scalability because the interconnection network would become a bottleneck. Similarly, while it is true that a non-

shared memory cluster can achieve good scalability, it does so only if one excludes the data distribution and collection phases of the program from the test. These times are trivial on an Altix system due to its shared memory architecture.

We have also run the STREAM benchmark without thread pinning, both with the standard 2.4.18 scheduler and the O(1) scheduler. The O(1) scheduler produces a throughput result that is nearly six times better than the standard Linux scheduler. This demonstrates that the standard Linux scheduler causes dramatically more thread migration and cache damage than the O(1) scheduler.

7.2 Star-CD™

Star-CD [21] is a fluid-flow analysis system marketed by Computational Dynamics, Ltd. It is an example of a computational fluid dynamics code. This particular code uses a message-passing interface on top of the shared-memory Altix hardware. This example uses an "A" Class Model, with 6 million cells. The results of the benchmark are shown in Figure 6.

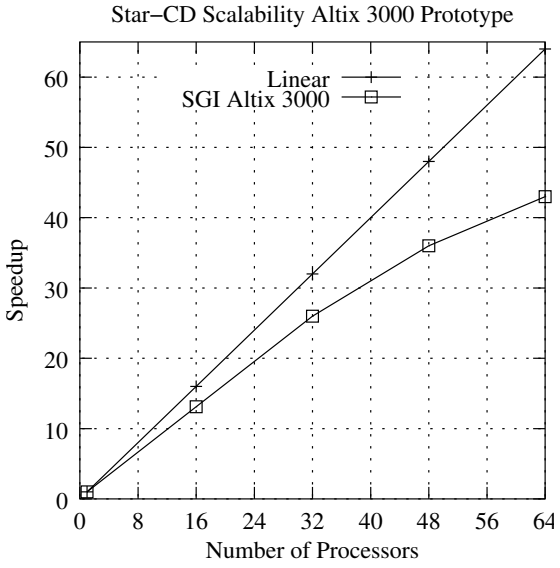

Figure 6: Scalability of Star-CD Benchmark

7.3 Gaussian® 98

Gaussian [9] is a product of Gaussian, Inc. Gaussian is a computational chemistry system that calculates molecular properties based on fundamental quantum mechanics principles. The problem being solved in this case is a sample problem from the Gaussian quality assurance test suite. This code uses a shared memory programming model. The results of the benchmark are shown in Figure 7.

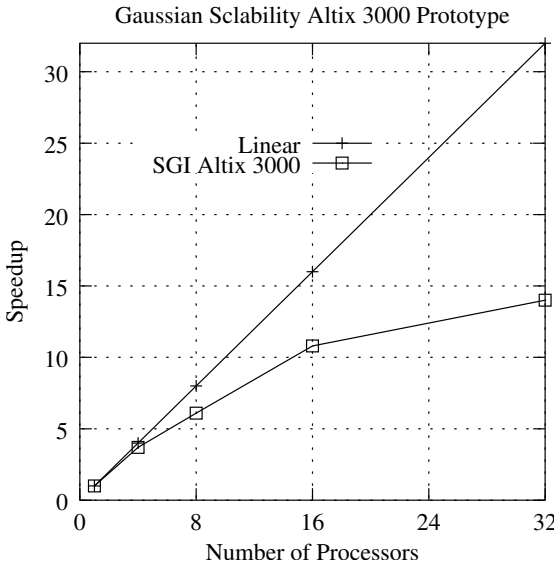

Figure 7: Scalability of Gaussian Benchmark

8 Concluding Remarks

With the open-source changes discussed in this paper, SGI has found Linux to be a good fit

for HPC applications. In particular, changes in the Linux scheduler, use of the XFS file system, use of the XSCSI implementation, and numerous small scalability fixes have significantly improved scaling of Linux for the Altix platform. The combination of a relatively low remote-to-local memory access time ratio and the high bandwidth provided by the Altix hardware are also key reasons that we have been able to achieve good scalability using Linux on the Altix system.

Relatively speaking, the total set of changes in Linux for the Altix system is small, and most of the changes have been obtained from the open-source Linux community. SGI is committed to working with the Linux community to ensure that Linux performance and scalability continues to improve as a viable and competitive operating system for real-world environments. Many of the changes discussed in this paper have already been submitted to the open-source community for inclusion in community maintained software. Versions of this software are also available at `oss.sgi.com`.

We anticipate future scalability efforts in multiple directions. One is an ongoing reduction of lock contention, as best we can accomplish with the 2.4 source base. We have work in progress on CPU scheduler improvements, reduction of the *pagecache_lock* contention, and reductions in other I/O-related locks.

Another class of work will be analysis of cache and TLB activity inside the kernel, which will presumably generate patches to reduce overhead associated with poor cache and TLB usage.

Large-scale I/O is another area of ongoing focus, including improving aggregate bandwidth, increasing transaction rates, and spreading interrupt handlers across multiple CPUs.

References

[1] `sourceforge.net/projects/discontig`

[2] `www.kernel.org/pub/linux/kernel/people/mbligh/`

[3] Ray Bryant and Bill Hartner, Java technology, threads, and scheduling in Linux, *IBM Developerworks* `www-106.ibm.com/developerworks/library/j-java2/index.html`

[4] Ray Bryant and John Hawkes, Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel, *Proceedings of the Fourth Annual Linux Showcase & Conference*, Atlanta, Ga. (2000), `oss.sgi.com/projects/lockmeter/als2000/als2000lock.html`

[5] Ray Bryant, Ruth Forester, and John Hawkes, Filesystem Performance and Scalability in Linux 2.4.17, *Proceedings of the Freenix Track of the 2002 Usenix Annual Technical Conference*, Montery, Ca., (June 2002).

[6] Ray Bryant, David Raddatz, and Roger Sunshine, PenguinoMeter: A new File-I/O Benchmark for Linux, *Proceedings of the 5th Annual Linux Showcase and Conference*, Oakland, Ca. (October 2001).

[7] `home.arcor.de/efocht/sched`

[8] M. Kravetz, H. Franke, S. Nagar, and R. Ravindran, Enhancing Linux Scheduler Scalability, *Proceedings of the Ottawa Linux Symposium, Ottawa, CA*, July 2001.

[9] Gaussian, Inc., Carnegie Office Park, Building 6, Suite 230, Carnegie, PA 15106 USA. `www.guassian.com`

[10] James Laudon and Daniel Lenoski, The SGI Origin: a ccNUMA Highly Scalable Server, *ACM SIGARCH Computer Architecture News*, Volume 25, Issue 2, (May 1997), pp. 241-251.

[11] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennesy, The DASH prototype: Logic overhead and performance, *IEEE Transacctions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[12] John D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, `www.cs.virginia.edu/stream`

[13] David Mosberger and Stephane Eranian, *IA-64 Linux Kernel, Design and Implementation*, Prentice-Hall (2002), ISBN 0-13-061014-3, pp. 405-406.

[14] `www.kernel.org/pub/linux/kernel/ports/ia64`

[15] `www.hpl.hp.com/research/linux/perfmon/`.

[16] `www.ussg.iu.edu/hypermail/linux/kernel/0201.0/0810.html`

[17] `oss.sgi.com/projects/lockmeter`

[18] `oss.sgi.com/projects/kernprof`

[19] `lse.sourceforge.net/lockhier/bkl_rollup.html`

[20] `www.intel.com/software/products/vtune`

[21] Star-CD, `www.cd-adapco.com`.

[22] `www.spec.org`

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*