Scaling Linux® to the Extreme

From 64 to 512 Processors

Ray Bryant raybry@sgi.com
John Hawkes hawkes@sgi.com
Jack Steiner steiner@sgi.com
Jesse Barnes jbarnes@sgi.com
Jeremy Higdon jeremy@sgi.com

Silicon Graphics, Inc.

Abstract

In January 2003, SGI announced the SGI® Altix® 3000 family of servers. As announced, the SGI Altix 3000 system supported up to 64 Intel® Itanium® 2 processors and 512 GB of main memory in a single Linux® image. Altix now supports up to 256 processors in a single Linux system, and we have a few early-adopter customers who are running 512 processors in a single Linux system; others are running with as much as 4 terabytes of memory. This paper continues the work reported on in our 2003 OLS paper by describing the changes necessary to get Linux to efficiently run high-performance computing workloads on such large systems.

Introduction

At OLS 2003 [1], we discussed changes to Linux that allowed us to make Linux scale to 64 processors for our high-performance computing (HPC) workloads. Since then, we have continued our scalability work, and we now support up to 256 processors in a single Linux image, and we have a few early-adopter customers who are running 512 processors in a single-system image; other customers are running with as much as 4 terabytes of memory.

As can be imagined, the type of changes necessary to get a single Linux system to scale on a 512 processor system or to support 4 terabytes of memory are of a different nature than those necessary to get Linux to scale up to a 64 processor system, and the majority of this paper will describe such changes.

While much of this work has been done in the context of a Linux 2.4 kernel, Altix is now a supported platform in the Linux 2.6 series (`www.kernel.org` versions of Linux 2.6 boot and run well on many small to moderate sized Altix systems), and our plan is to port many of these changes to Linux 2.6 and propose them as enhancements to the community kernel. While some of these changes will be unique to the Linux kernel for Altix, many of the changes we propose will also improve performance on smaller SMP and NUMA systems, so should be of general interest to the Linux scalability community.

In the rest of this paper, we will first provide a brief review of the SGI Altix 3000 hardware. Next we will describe why we believe that very large single-system image, shared-memory machine can be more effective tools for HPC than similar sized non-shared memory clusters. We will then discuss changes that we made to Linux for Altix in order to make

that system a more effective system for HPC on systems with as many as 512 processors. A second large topic of discussion will be the changes to support high-performance I/O on Altix and some of the hardware underpinnings for that support. We believe that the latter set of problems are general in the sense that they apply to any large scale NUMA system and the solutions we have adopted should be of general interest for this reason.

Even though this paper is focused on the changes that we have made to Linux to effectively support very large Altix platforms, it should be remembered that the total number of such changes is small in relation to the overall size of the Linux kernel and its supporting software. SGI is committed to supporting the Linux community and continues to support Linux for Altix as a member of the Linux family of kernels, and in general to support binary compatibility between Linux for Altix and Linux on other Itanium Processor Family platforms.

In many cases, the scaling changes described in this paper have already been submitted to the community for consideration for inclusion in Linux 2.6. In other cases, the changes are under evaluation to determine if they need to be added to Linux 2.6, or whether they are fixes for problems in Linux 2.4.21 (the current product base for Linux for Altix) that are no longer present in Linux 2.6.

Finally, this paper contains forward-looking statements regarding SGI® technologies and third-party technologies that are subject to risks and uncertainties. The reader is cautioned not to rely unduly on these forward-looking statements, which are not a guarantee of future or current performance, nor are they a guarantee that features described herein will or will not be available in future SGI products.

The SGI Altix Hardware

This section is condensed from [1]; the reader should refer to that paper for additional details.

An Altix system consists of a configurable number of rack-mounted units, each of which SGI refers to as a *brick*. The most common type of brick is the C-brick (or compute brick). A fully configured C-brick consists of two separate dual-processor Intel Itanium 2 systems, each of which is a bus-connected multiprocessor or *node*.

In addition to the two processors on the bus, there is also a SHUB chip on each bus. The SHUB is a proprietary ASIC that (1) acts as a memory controller for the local memory, (2) provides the interface to the interconnection network, (3) manages the global cache coherency protocol, and (4) some other functions as discussed in [1].

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. The SHUB detects whether a reference is local, in which case it directs the request to the memory on the node, or remote, in which case it forwards the request across the interconnection network to the SHUB chip where the memory reference will be serviced.

Local memory references have lower latency; the Altix system is thus a NUMA (non-uniform 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.

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 local

bus and is used to maintain cache-coherency between the Itanium processors on the bus. 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 [2].

The Altix system interconnection network uses routing bricks to provide connectivity in system sizes larger than 16 processors. In systems with 128 or more processors a second layer of routing bricks is used to forward requests among subgroups of 32 processors each. The routing topology is a fat-tree topology with additional "express" links being inserted to improve performance.

Why Big SSI?

In this section we discuss the rationale for building such a large single-system image (SSI) box as an Altix system with 512 CPU's and (potentially) several TB of main memory:

(1) Shared memory systems are more flexible and easier to manage than a cluster. One can simulate message passing on shared memory, but not the other way around. Software for cluster management and system maintenance exists, but can be expensive or complex to use.

(2) Shared memory style programming is generally simpler and more easily understood than message passing. Debugging of code is often simpler on a SSI system than on a cluster.

(3) It is generally easier to port or write codes from scratch using the shared memory paradigm. Additionally it is often possible to simply ignore large sections of the code (e.g. those devoted to data input and output) and only parallelize the part that matters.

(4) A shared memory system supports easier load balancing within a computation. The mapping of grid points to a node determines the computational load on the node. Some grid points may be located near more rapidly changing parts of computation, resulting in higher computational load. Balancing this over time requires moving grid points from node to node in a cluster, where in a shared memory system such re-balancing is typically simpler.

(5) Access to large global data sets is simplified. Often, the parallel computation depends on a large data set describing, for example, the precise dimensions and characteristics of the physical object that is being modeled. This data set can be too large to fit into the node memories available on a clustered machine, but it can readily be loaded into memory on a large shared memory machine.

(6) Not everything fits into the cluster model. While many production codes have been converted to message passing, the overall computation may still contain one or more phases that are better performed using a large shared memory system. Or, there may be a subset of users of the system who would prefer a shared memory paradigm to a message passing one. This can be a particularly important consideration in large data-center environments.

Kernel Changes

In this section we describe the most significant kernel problems we have encountered in running Linux on a 512 processor Altix system.

Cache line and TLB Conflicts

Cache line conflicts occur in every cache-coherent multiprocessor system, to one extent or another, and whether or not the conflict exhibits itself as a performance problem is dependent on the rate at which the conflict occurs and the time required by the hardware to resolve

the conflict. The latter time is typically proportional to the number of processors involved in the conflict. On Altix systems with 256 processors or more, we have encountered some cache line conflicts that can effectively halt forward progress of the machine. Typically, these conflicts involve global variables that are updated at each timer tick (or faster) by every processor in the system.

One example of this kind of problem is the default kernel profiler. When we first enabled the default kernel profiler on a 512 CPU system, the system would not boot. The reason was that once per timer tick, each processor in the system was trying to update the profiler bin corresponding to the CPU idle routine. A work around to this problem was to initialize `prof_cpu_mask` to `CPU_MASK_NONE` instead of the default. This disables profiling on all processors until the user sets the `prof_cpu_mask`.

Another example of this kind of problem was when we imported some timer code from Red Hat® AS 3.0. The timer code included a global variable that was used to account for differences between HZ (typically a power of 2) and the number of microseconds in a second (nominally 1,000,000). This global variable was updated by each processor on each timer tick. The result was that on Altix systems larger than about 384 processors, forward progress could not be made with this version of the code. To fix this problem, we made this global variable a per processor variable. The result was that the adjustment for the difference between HZ and microseconds is done on a per processor rather than on a global basis, and now the system will boot.

Still other cache line conflicts were remedied by identifying cases of false cache line sharing i.e., those cache lines that inadvertently contain a field that is frequently written by one CPU and another field (or fields) that are frequently read by other CPUs.

Another significant bottleneck is the ia64 `do_gettimeofday()` with its use of `cmpxchg`. That operation is expensive on most architectures, and concurrent `cmpxchg` operations on a common memory location scale worse than concurrent simple writes from multiple CPUs. On Altix, four concurrent user `gettimeofday()` system calls complete in almost an order of magnitude more time than a single `gettimeofday()`; eight are 20 times slower than one; and the scaling deteriorates nonlinearly to the point where 32 concurrent system calls is 100 times slower than one. At the present time, we are still exploring a way to improve this scaling problem in Linux 2.6 for Altix.

While moving data to per-processor storage is often a solution to the kind of scaling problems we have discussed here, it is not a panacea, particularly as the number of processors becomes large. Often, the system will want to inspect some data item in the per-processor storage of each processor in the system. For small numbers of processors this is not a problem. But when there are hundreds of processors involved, such loops can cause a TLB miss each time through the loop as well as a couple of cache-line misses, with the result that the loop may run quite slowly. (A TLB miss is caused because the per-processor storage areas are typically isolated from one another in the kernel's virtual address space.)

If such loops turn out to be bottlenecks, then what one must often do is to move the fields that such loops inspect out of the per-processor storage areas, and move them into a global static array with one entry per CPU.

An example of this kind of problem in Linux 2.6 for Altix is the current allocation scheme of the per-CPU run queue structures. Each

per-CPU structure on an Altix system requires a unique TLB to address it, and each structure begins at the same virtual offset in a page, which for a virtually indexed cache means that the same fields will collide at the same index. Thus, a CPU scheduler that wishes to do a quick peek at every other CPU's `nr_running` or `cpu_load` will not only suffer a TLB miss on every access, but will also likely suffer a cache miss because these same virtual offsets will collide in the cache. Cache coloring of these addresses would be one way to solve this problem; we are still exploring ways to fix this problem in Linux 2.6 for Altix.

Lock Conflicts

A cousin of cache line conflicts are the lock conflicts. Indeed, the root mechanism of the lock bottleneck is a cache line conflict. For a `spinlock_t` the conflict is the `cmpxchg` operation on the word that signifies whether or not the lock is owned. For a `rwlock_t` the conflict is the cmpxchg or fetch-and-add operation on the count of the number of readers or the bit signifying whether or not the lock is owned exclusively by a writer. For a `seqlock_t` the conflict is the increment of the sequence number.

For some lock conflicts, such as the `rcu_ctrlblk.mutex`, the remedy is to make the spinlock more fine-grained, e.g., by making it hierarchical or per-CPU. For other lock conflicts, the most effective remedy is to reduce the use of the lock.

The O(1) CPU scheduler replaced the global `runqueue_lock` with per-CPU run queue locks, and replaced the global run queue with per-CPU run queues. While this did substantially decrease the CPU scheduling bottleneck for CPU counts in the 8 to 32 range, additional effort has been necessary to remedy additional bottlenecks that appear with even large configurations.

For example, we discovered that at 256 processors and above, we encountered a live lock early in system boot because hundreds of idle CPUs are load-balancing and are racing in contention on one or a few busy CPUs. The contention is so severe that the busy CPU's scheduler cannot itself acquire its own run queue lock, and thus the system live locks.

A remedy we applied in our Altix 2.4-based kernel was to introduce a progressively longer back off between successive load-balancing attempts, if the load-balancing CPU continues to be unsuccessful in finding a task to pull-migrate. Perhaps all the busiest CPU's tasks are pinned to that CPU, or perhaps all the tasks are still cache-hot. Regardless of the reason, a load-balancing failure results in that CPU delaying the next load-balance attempt by another incremental increase in time. This algorithm effectively solved the live lock, as well as improved other high-contention conflicts on a busiest CPU's run queue lock (e.g., always finding pinned tasks that can never be migrated).

This load-balance back off algorithm did not get accepted into the early 2.6 kernels. The latest 2.6.7 CPU scheduler, as developed by Nick Piggin, incorporates a similar back off algorithm. However, this algorithm (at least as it appears in 2.6.7-rc2) continues to cause a boot-time live lock at 512 processors on Altix so we are continuing to investigate this matter.

Page Cache

Managing the page cache in Altix has been a challenging problem. The reason is that while a large Altix system may have a lot of memory, each node in the system only has a relatively small fraction of that memory available as local memory. For example, on a 512 CPU sys-

tem, if the entire system has 512 GB of memory, each node on the system has only 2 GB of local memory; less than 0.4% of the available memory on the system is local. When you consider that it is quite common on such systems to deal with files that are tens of GB in size, it is easy to understand how the page cache could consume all of the memory on several nodes in the system just doing normal, buffered-file I/O.

Stated another way, this is the challenge of a large NUMA system: all memory is addressable, but only a tiny fraction of that memory is local. Users of NUMA systems need to place their most frequently accessed data in local memory; this is crucial to obtain the maximum performance possible from the system. Typically this is done by allocating pages on a first-touch basis; that is, we attempt to allocate a page on the node where it is first referenced. If all of the local memory on a node is consumed by the page cache, then these local storage allocations will spill over to other (remote) nodes, the result being a potentially significant impact on program performance.

Similarly, it is important that the amount of free memory be balanced across idle nodes in the system. An imbalance could lead to some components of a parallel computation running slower than others because not all components of the computation were able to allocate their memory entirely out of local storage. Since the overall speed of parallel computation is determined by the execution of its slowest component, the performance of the entire application can be impacted by a non-local storage allocation on only a few nodes.

One might think that `bdflush` or `kupdated` (in a Linux 2.4 system) would be responsible for cleaning up unused page-cache pages. As the OLS reader knows, these daemons are responsible not for deallocating page-cache pages, but cleaning them. It is the swap daemon `kswapd` that is responsible for causing page-cache pages to be deallocated. However, in many situations we have encountered, even though multiple nodes of the system would be completely out of local memory, there would still be lots of free memory elsewhere in the system. As a result, `kswapd` will never start. Once the system gets into such a state, the local memory on those nodes can remain allocated entirely to page-cache pages for very long stretches of time since as far as the kernel is concerned there is no memory "pressure". To get around this problem, particularly for benchmarking studies, users have often resorted to programs that allocate and touch all of the memory on the system, thus causing `kswapd` to wake up and free unneeded buffer cache pages.

We have dealt with this problem in a number of ways, but the first approach was to change `page_cache_alloc()` so that instead of allocating the page on the local node, we spread allocations across all nodes in the system. To do this, we added a new GFP flag: `GFP_ROUND_ROBIN` and a new procedure: `alloc_pages_round_robin()`. `alloc_pages_round_robin()` maintains a counter in per-CPU storage; the counter is incremented on each call to `page_cache_alloc()`. The value of the counter, modulus the number of nodes in the system, is used to select the `zonelist` passed to `__alloc_pages()`. Like other NUMA implementations, in Linux for Altix there is a `zonelist` for each node, and the `zonelist`s are sorted in nearest neighbor order with the `zone` for the local node as the first entry of the `zonelist`. The result is that each time `page_cache_alloc()` is called, the returned page is allocated on the next node in sequence, or as close as possible to that node.

The rationale for allocating page-cache pages

in this way is that while pages are local resources, the page cache is a global resource, usable by all processes on the system. Thus, even if a process is bound to a particular node, in general it does not make sense to allocate page-cache pages just on that node, since some other process in the system may be reading that same file and hence sharing the pages. So instead of flooding the current node with the page-cache pages for files that processes on that node have opened, we "tax" every node in the system with a fraction of the page-cache pages. In this way, we try to conserve a scarce resource (local memory) by spreading page-cache allocations over all nodes in the system.

However, even this step was not enough to keep local storage usage balanced among nodes in the system. After reading a 10 GB file, for example, we found that the node where the reading process was running would have up to 40,000 pages more storage allocated than other nodes in the system. It turned out the reason for this was that buffer heads for the read operation were being allocated locally. To solve this problem in our Linux 2.4.21 kernel for Altix, we modified `kmem_cache_grow()` so that it would pass the `GFP_ROUND_ROBIN` flag to `kmem_getpages()` with the result that the slab caches on our systems are now also allocated out of round-robin storage. Of course, this is not a perfect solution, since there are situations where it makes perfect sense to allocate a slab cache entry locally; but this was an expedient solution appropriate for our product. For Linux 2.6 for Altix we would like to see the slab allocator be made NUMA aware. (Manfred Spraul has created some patches to do this and we are currently evaluating these changes.)

The previous two changes solved many of the cases where a local storage could be exhausted by allocation of page-cache pages. However, they still did not solve the problem of local allocations spilling off node, particularly in those cases where storage allocation was tight across the entire system. In such situations, the system would often start running the synchronous swapping code even though most (if not all) of the page-cache pages on the system were clean and unreferenced outside of the page-cache. With the very-large memory sizes typical of our larger Altix customers, entering the synchronous swapping code needs to be avoided if at all possible since this tends to freeze the system for 10s of seconds. Additionally, the round robin allocation fixes did not solve the problem of poor and unrepeatable performance on benchmarks due to the existence of significant amounts of page-cache storage left over from previous executions.

To solve these problems, we introduced a routine called `toss_buffer_cache_pages_node()` (referred to here as `toss()`, for brevity). In a related change, we made the active and inactive lists per node rather than global. `toss()` first scans the inactive list (on a particular node) looking for idle page-cache pages to release back to the free page pool. If not enough such pages are found on the inactive list, then the active list is also scanned. Finally, if `toss()` has not called `shrink_slab_caches()` recently, that routine is also invoked in order to more aggressively free unused slab-cache entries. `toss()` was patterned after the main loop of `shrink_caches()` except that it would never call `swap_out()` and if it encountered a page that didn't look to be easily free able, it would just skip that page and go on to the next page.

A call to `toss()` was added in `__alloc_pages()` in such a way that if allocation on the current node fails, then before trying to allocate from some other node (i. e. spilling to another node), the system will first see if it can free enough page-cache pages from the current node so that the current node alloca-

tion can succeed. In subsequent allocation passes, `toss()` is also called to free page-cache pages on nodes other than the current one. The result of this change is that clean page-cache pages are effectively treated as free memory by the page allocator.

At the same time that the `toss()` code was added, we added a new user command `bcfree` that could be used to free all idle page-cache pages. (On the `__alloc_pages()` path, `toss()` would only try to free 32 pages per node.) The `bcfree` command was intended to be used only for resetting the state of the page cache before running a benchmark, and in lieu of rebooting the system in order to get a clean system state. However, our customers found that this command could be used to reduce the size of the page cache and to avoid situations where large amounts of buffered-file I/O could force the system to begin swapping. Since `bcfree` kills the entire page-cache, however, this was regarded as a substandard solution that could also hurt read performance of cached data and we began looking for another way to solve this "BIGIO" problem.

Just to be specific, the BIGIO problem we were trying to solve was based on the behavior of our Linux 2.4.21 kernel for Altix. A customer reported that on a 256 GB Altix system, if 200 GB were allocated and 50 GB free, that if the user program then tried to write 100 GB of data out to disk, the system would start to swap, and then in many cases fill up the swap space. At that point our Out-of-memory (OOM) killer would wake up and kill the user program! (See the next section for discussion of our OOM killer changes.)

Initially we were able to work around this problem by increasing the amount of swap space on the system. Our experiments showed that with an amount of swap space equal to one-quarter the main memory size, the 256 GB example discussed above would continue to completion without the OOM killer being invoked. I/O performance during this phase was typically one-half of what the hardware could deliver, since two I/O operations often had to be completed: one to read the data in from the swap device, and one to write the data to the output file. Additionally, while the swap scan was active, the system was very sluggish. These problems led us to search for another solution.

Eventually what we developed is an aggressive method of trimming the page cache when it started to grow too big. This solution involved several steps:

(1) We first added a new page list, the `reclaim_list`. This increased the size of `struct page` by another 16 bytes. On our system, `struct page` is allocated on cache-aligned boundaries anyway, so this really did not cause an increase in storage, since the current `struct page` size was less than 112 bytes. Pages were added to the reclaim list when they were inserted into the page cache. The reclaim list is per node, with per node locking. Pages were removed from the reclaim list when they were no longer reclaimable; that is, they were removed from the reclaim list when they were marked as dirty due to buffer file-I/O or when they were mapped into an address space.

(2) We rewrote `toss()` to scan the reclaim list instead of the inactive and active lists. Herein we will refer to the new version of `toss()` as `toss_fast()`.

(3) We introduced a variant of `page_cache_alloc()` called `page_cache_alloc_limited()`. Associated with this new routine were two control variables settable via `sysctl()`: `page_cache_limit` and `page_cache_limit_threshold`.

(4) We modified the `generic_file_write()` path to call `page_cache_alloc_limited()` instead of `page_cache_alloc()`. `page_cache_alloc_limited()` examines the size of the page cache. If the total amount of free memory in the system is less than `page_cache_limit_threshold` and the size of the page cache is larger than `page_cache_limit`, then `page_cache_alloc_limited()` calls `page_cache_reduce()` to free enough page-cache pages on the system to bring the page cache size down below `page_cache_limit`. If this succeeds, then `page_cache_alloc_limited()` calls `page_cache_alloc` to allocate the page. If not, then we wakeup `bdflush` and the current thread is put to sleep for 30ms (a tunable parameter)

The rationale for the `reclaim_list` and `toss_fast()` was that when we needed to trim the page cache, practically all pages in the system would typically be on the inactive list. The existing `toss()` routine scanned the inactive list and thus was too slow to call from `generic_file_write`. Moreover, most of the pages on the inactive list were not reclaimable anyway. Most of the pages on the `reclaim_list` are reclaimable. As a result `toss_fast()` runs much faster and is more efficient at releasing idle page-cache pages than the old routine.

The rationale for the `page_cache_limit_threshold` in addition to the `page_cache_limit` is that if there is lots of free memory then there is no reason to trim the page cache. One might think that because we only trim the page cache on the file write path that this approach would still let the page cache to grow arbitrarily due to file reads. Unfortunately, this is not the case, since the Linux kernel in normal multiuser operation is constantly writing something to the disk. So, a page cache limit enforced at file write time is also an effective limit on the size of the page cache due to file reads.

Finally, the rationale for delaying the calling task when `page_cache_reduce()` fails is that we do not want the system to start swapping to make space for new buffered I/O pages, since that will reduce I/O bandwidth by as much as one-half anyway, as well as take a lot of CPU time to figure out which pages to swap out. So it is better to reduce the I/O bandwidth directly, by limiting the rate of requested I/O, instead of allowing that I/O to proceed at rate that causes the system to be overrun by page-cache pages.

Thus far, we have had good experience with this algorithm. File I/O rates are not substantially reduced from what the hardware can provide, the system does not start swapping, and the system remains responsive and usable during the period of time when the BIGIO is running.

Of course, this entire discussion is specific to Linux 2.4.21. For Linux 2.6, we have plans to evaluate whether this is a problem in the system at all. In particular, we want to see if an appropriate setting for `vm_swappiness` to zero can eliminate the "BIGIO causes swapping" problem. We also are interested in evaluating the recent set of VM patches that Nick Piggin [6] has assembled to see if they eliminate this problem for systems of the size of a large Altix.

VM and Memory Allocation Fixes

In addition to the page-cache changes described in the last section, we have made a number of smaller changes related to virtual memory and paging performance.

One set of such changes increased the parallelism of page-fault handling for anonymous

pages in multi-threaded applications. These applications allocate space using routines that eventually call `mmap()`; the result is that when the application touches the data area for the first time, it causes a minor page fault. These faults are serviced while holding the address space's `page_table_lock`. If the address space is large and there are a large number of threads executing in the address space, this spinlock can be an initialization-time bottleneck for the application. Examination of the `handle_mm_fault()` path for this case shows that the `page_table_lock` is acquired unconditionally but then released as soon as we have determined that this is a not-present fault for an anonymous page. So, we reordered the code checks in `handle_mm_fault()` to determine in advance whether or not this was the case we were in, and if so, to skip acquiring the lock altogether.

The second place the `page_table_lock` was used on this path was in `do_anonymous_page()`. Here, the lock was re-acquired to make sure that the process of allocating a page frame and filling in the pte is atomic. On Itanium, stores to page-table entries are normal stores (that is, the `set_pte` macro evaluates to a simple store). Thus, we can use `cmpxchg` to update the pte and make sure that only one thread allocates the page and fills in the pte. The compare and exchange effectively lets us lock on each individual pte. So, for Altix, we have been able to completely eliminate the `page_table_lock` from this particular page-fault path.

The performance improvement from this change is shown in Figure 1. Here we show the time required to initially touch 96 GB of data. As additional processors are added to the problem, the time required for both the baseline-Linux and Linux for Altix versions decrease until around 16 processors. At that point the `page_table_lock` starts to become a significant bottleneck. For the largest number of processors, even the time for the Linux for Altix case is starting to increase again. We believe that this is due to contention for the address space's `mmap` semaphore.

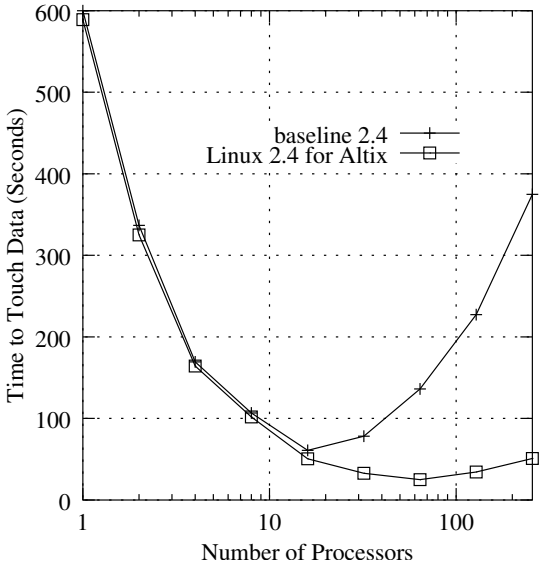

Figure 1: *Time to initially touch 96 GB of data.*

This is particularly important for HPC applications since OpenMP™[5], a common parallel programming model for FORTRAN, is implemented using a single address space, multiple-thread programming model. The optimization described here is one of the reasons that Altix has recently set new performance records for the SPEC® SPEComp® L2001 benchmark [7].

While the above measurements were taken using Linux 2.4.21 for Altix, a similar problem exists in Linux 2.6. For many other architectures, this same kind of change can be made; i386 is one of the exceptions to this statement. We are planning on porting our Linux 2.4.21 based changes to Linux 2.6 and submitting the changes to the Linux community for inclusion in Linux 2.6. This may require moving part of `do_anonymous_page()` to architecture

dependent code to allow for the fact that not all architectures can use the compare and exchange approach to eliminate the use of the `page_table_lock` in `do_anonymous_page()`. However, the performance improvement shown in Figure 1 is significant for Altix so we would we would like to explore some way of incorporating this code into the mainline kernel.

We have encountered similar scalability limitations for other kinds of page-fault behavior. Figure 2 shows the number of page faults per second of wall clock time measured for multiple processes running simultaneously and faulting in a 1 GB `/dev/zero` mapping. Unlike the previous case described here, in this case each process has its own private mapping. (Here the number of processes is equal to the number of CPUs.) The dramatic difference between the baseline 2.4 and 2.6 cases and Linux for Altix is due to elimination of a lock in the super block for `/dev/zero`.

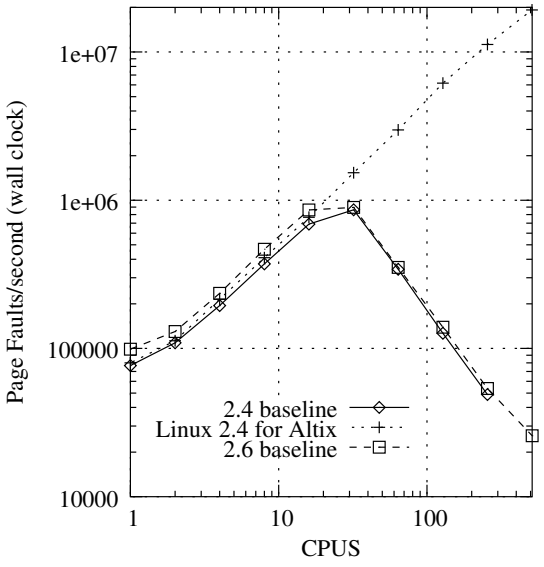

Figure 2: *Page Faults per Second of Wall Clock Time.*

The lock in the super block protects two counts: One count limits the maximum number of `/dev/zero` mappings to 2^{63}; the second count limits the number of pages assigned to a `/dev/zero` mapping to 2^{63}. Neither one of these counts is particularly useful for a `/dev/zero` mapping. We eliminated this lock and obtained a dramatic performance improvement for this micro-benchmark (at 512 CPUs the improvement was in excess of 800x). This optimization is important in decreasing startup time for large message-passing applications on the Altix system.

A related change is to distribute the count of pages in the page cache from a single global variable to a per node variable. Because every processor in the system needs to update the page-cache count when adding or removing pages from the page cache, contention for the cache line containing this global variable becomes significant. We changed this global count to a per-node count. When a page is inserted into (or removed from) the page cache, we update the page cache-count on the same node as the page itself. When we need the total number of pages in the page cache (for example if someone reads `/proc/meminfo`) we run a loop that sums the per node counts. However, since the latter operation is much less frequent than insertions and deletions from the page cache, this optimization is an overall performance improvement.

Another change we have made in the VM subsystem is in the out-of-memory (OOM) killer for Altix. In Linux 2.4.21, the OOM killer is called from the top of memory-free and swap-out call chain. `oom_kill()` is called from `try_to_free_pages_zone()` when calls to `shrink_caches()` at memory priority levels 6 through 0 have all failed. Inside `oom_kill()` a number of checks are performed, and if any of these checks succeed, the system is declared to not be out-of-memory. One of those checks is "if it has been more than 5 seconds since `oom_kill()` was last called, then we are not

OOM." On a large-memory Altix system, it can easily take much longer than that to complete the necessary calls to `shrink_caches()`. The result is that an Altix system never goes OOM in spite of the fact that swap space is full and there is no memory to be allocated.

It seemed to us that part of the problem here is the amount of time it can take for a swap full condition (readily detectable in `try_to_swap_out()` to bubble all the way up to the top level in `try_to_free_pages_zone()`, especially on a large memory machine. To solve this problem on Altix, we decided to drive the OOM killer directly off of detection of swap-space-full condition provided that the system also continues to try to swap out additional pages. A count of the number of successful swaps and unsuccessful swap attempts is maintained in `try_to_swap_out()`. If, in a 10 second interval, the number of successful swap outs is less than one percent of the number of attempted swap outs, and the total number of swap out attempts exceeds a specified threshold, then `try_to_swap_out()`) will directly wake the OOM killer thread (also new in our implementation). This thread will wait another 10 seconds, and if the out-of-swap condition persists, it will invoke `oom_kill()` to select a victim and kill it. The OOM killer thread will repeat this sleep and kill cycle until it appears that swap space is no longer full or the number of attempts to swap out new pages (since the thread went to sleep) falls below the threshold.

In our experience, this has made invocation of the OOM killer much more reliable than it was before, at least on Altix. Once again, this implementation was for Linux 2.4.21; we are in the process of evaluating this problem and the associated fix on Linux 2.6 at the present time.

Another fix we have made to the VM system in Linux 2.4.21 for Altix is in handling of HUGETLB pages. The existing implementation in Linux 2.4.21 allocates HUGETLB pages to an address space at `mmap()` time (see `hugetlb_prefault()`); it also zeroes the pages at this time. This processing is done by the thread that makes the `mmap()` call. In particular, this means that zeroing of the allocated HUGETLB pages is done by a single processor. On a machine with 4 TB of memory and with as much memory allocated to HUGETLB pages as possible, our measurements have shown that it can take as long as 5,000 seconds to allocate and zero all available HUGETLB pages. Worse yet, the thread that does this operation holds the address space's `mmap_sem` and the `page_table_lock` for the entire 5,000 seconds. Unfortunately, many commands that query system state (such as `ps` and `w`) also wish to acquire one of these locks. The result is that the system appears to be hung for the entire 5,000 seconds.

We solved this problem on Altix by changing the implementation of HUGETLB page allocation from *prefault* to *allocate on fault*. Many others have created similar patches; our patch was unique in that it also allowed zeroing of pages to occur in parallel if the HUGETLB page faults occurred on different processors. This was crucial to allow a large HUGETLB page region to be faulted into an address space in parallel, using as many processors as possible. For example, we have observed speedups of 25x using 16 processors to touch O(100 GB) of HUGETLB pages. (The speedup is super linear because if you use just one processor it has to zero many remote pages, whereas if you use more processors, at least some of the pages you are zeroing are local or on nearby nodes.) Assuming we can achieve the same kind of speedup on a 4 TB system, we would reduce the 5,000 second time stated above to 200 seconds.

Recently, we have worked with Kenneth Chen

to get a similar set of changes proposed for Linux 2.6 [3]. Once this set of changes is accepted into the mainline this particular problem will be solved for Linux 2.6. These changes are also necessary for Andi Kleen's NUMA placement algorithms [4] to apply to HUGETLB pages, since otherwise pages are placed at `hugetlb_prefault()` time.

A final set of changes is related to large kernel tables. As previously mentioned, on an Altix system with 512 processors, less than 0.4% of the available memory is local. Certain tables in the Linux kernel are sized to be on the order of one percent of available memory. (An example of this is the TCP/IP hash table.) Allocating a table of this size can use all of the local memory on a node, resulting in exactly the kind of storage-allocation imbalance we developed the page-cache changes to solve. To avoid this problem, we also implement round-robin allocation of these large tables. Our current technique uses `vm_alloc()` to do this. Unfortunately, this is not portable across all architectures, since certain architectures have limited amounts of space that can be allocated by `vm_alloc()`. Nonetheless, this is a change that we need to make; we are still exploring ways of making this change acceptable to the Linux community.

Once we have solved the initial allocation problem for these tables, there is still the problem of getting them appropriately sized for an Altix system. Clearly if there are 4 TB of main memory, it does not make much sense to allocate a TCP/IP hash table of 40 GB, particularly since the TCP/IP traffic into an Altix system does not increase with memory size the way one might expect it to scale with a traditional Linux server. We have seen cases where system performance is significantly hampered due to lookups in these overly large tables. At the moment, we are still exploring a solution acceptable to the community to solve this particular problem.

I/O Changes for Altix

One of the design goals for the Altix system is that it support standard PCI devices and their associated Linux drivers as much as possible. In this section we discuss the performance improvements built into the Altix hardware and supported through new driver interfaces in Linux that help us to meet this goal with excellent performance even on very large Altix systems.

According to the PCI specification, DMA writes and PIO read responses are strongly ordered. On large NUMA systems, however, DMA writes can take a long time to complete. Since most PIO reads do not imply completion of a previous DMA write, relaxing the ordering rules of DMA writes and PIO read responses can greatly improve system performance.

Another large system issue relates to initiating PIO writes from multiple CPUs. PIO writes from two different CPUs may arrive out of order at a device. The usual way to ensure ordering is through a combination of locking and a PIO read (see Documentation/io_ordering.txt). On large systems, however, doing this read can be very expensive, particularly if it must be ordered with respect to unrelated DMA writes.

Finally, the NUMA nature of large machines make some optimizations obvious and desirable. Many devices use so-called consistent system memory for retrieving commands and storing status information; allocating that memory close to its associated device makes sense.

Making non–dependent PIO reads fast

In its I/O chipsets, SGI chose to relax the ordering between DMAs and PIOs, instead adding

a barrier attribute to certain DMA writes (to consistent PCI allocations on Altix) and to interrupts. This works well with controllers that use DMA writes to indicate command completions (for example a SCSI controller with a response queue, where the response queue is allocated using `pci_alloc_consistent`, so that writes to the response queue have the barrier attribute). When we ported Linux to Altix, this behavior became a problem, because many Linux PCI drivers use PIO read responses to imply a status of a DMA write. For example, on an IDE controller, a bit status register read is performed to find out if a command is complete (command complete status implies that DMA writes of that command's data are completed). As a result, SGI had to implement a rather heavyweight mechanism to guarantee ordering of DMA writes and PIO reads. This mechanism involves doing an explicit flush of DMA write data after each PIO read.

For the cases in which strong ordering of PIO read responses and DMA writes are not necessary, a new API was needed so that drivers could communicate that a given PIO read response could used relaxed ordering with respect to prior DMA writes. The `read_relaxed` API [8] was added early in the 2.6 series for this purpose, and mirrors the normal read routines, which have variants for various sized reads.

The results below show how expensive a normal PIO read transaction can be, especially on a system doing a lot of I/O (and thus DMA).

Type of PIO	Time (ns)
normal PIO read	3875
relaxed PIO read	1299

Table 1: Normal vs. relaxed PIO reads on an idle system

It remains to be seen whether this API will also apply to the newly added RO bit in the PCI-X specification—the author is hopeful! Either way, it does give hardware vendors who want to support Linux some additional flexibility in their design.

Type of PIO	Time (ns)
normal PIO read	4889
relaxed PIO read	1646

Table 2: Normal vs. relaxed PIO reads on a busy system

Ordering posted writes efficiently

On many platforms, PIO writes from different CPUs will not necessarily arrive in order (i.e., they may be intermixed) even when locking is used. Since the platform has no way of knowing whether a given PIO read depends on preceding writes, it has to guarantee that all writes have completed before allowing a read transaction to complete. So performing a read prior to releasing a lock protecting a region doing writes is sufficient to guarantee that the writes arrive in the correct order.

However, performing PIO reads can be an expensive operation, especially if the device is on a distant node. SGI chipset designers foresaw this problem, however, and provided a way to ensure ordering by simply reading a register from the chipset on the local node. When the register indicates that all PIO writes are complete, it means they have arrived at the chipset attached to the device, and so are guaranteed to arrive at the device in the intended order. The SGI sn2 specific portion of the Linux ia64 port (sn2 is the architecture name for Altix in the Linux kernel source tree) provides a small function, `sn_mmiob()` (for memory–mapped I/O barrier, analogous to the `mb()` macro), to do just that. It can be used in place of reads that are intended to deal with posted writes and provides some benefit:

Type of flush	Time (ns)
regular PIO read	5940
relaxed PIO read	2619
`sn_mmiob()`	1610
(local chipset read alone)	399

Table 3: Normal vs. fast flushing of 5 PIO writes

Adding this API to Linux (i.e., making it non-sn2-specific) was discussed some time ago [9], and may need to be raised again, since it does appear to be useful on Altix, and is probably similarly useful on other platforms.

Local allocation of consistent DMA mappings

Consistent DMA mappings are used frequently by drivers to store command and status buffers. They are frequently read and written by the device that owns them, so making sure they can be accessed quickly is important. The table below shows the difference in the number of operations per second that can be achieved using local versus remote allocation of consistent DMA buffers. Local allocations were guaranteed by changing the `pci_alloc_consistent` function so that it calls `alloc_pages_node` using the node closest to the PCI device in question.

Type	I/Os per second
Local consistent buffer	46231
Remote consistent buffer	41295

Table 4: Local vs. remote DMA buffer allocation

Although this change is platform specific, it can be made generic if a `pci_to_node` or `pci_to_nodemask` routine is added to the Linux topology API.

Concluding Remarks

Today, our Linux 2.4.21 kernel for Altix provides a productive platform for our high-performance-computing users who desire to exploit the features of the SGI Altix 3000 hardware. To achieve this goal, we have made a number of changes to our Linux for Altix kernel. We are now in the process of either moving those changes forward to Linux 2.6 for Altix, or of evaluating the Linux 2.6 kernel on Altix in order to determine if these changes are indeed needed at all. Our goal is to develop a version of the Linux 2.6 kernel for Altix that not only supports our HPC customers equally well as our existing Linux 2.4.21 kernel, but also consists as much as possible of community supported code.

References

[1] Ray Bryant and John Hawkes, Linux Scalability for Large NUMA Systems, *Proceedings of the 2003 Ottawa Linux Symposium*, Ottawa, Ontario, Canada, (July 2003).

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

[3] Kenneth Chen, "hugetlb demand paging patch part [0/3]," linux-kernel@vger.kernel.org, 2004-04-13 23:17:04, `http://marc.theaimsgroup.com/?l=linux-kernel&m=108189860419356&w=2`

[4] Andi Kleen, "Patch: NUMA API for Linux," linux-kernel@vger.kernel.org,

Tue, 6 Apr 2004 15:33:22 +0200, `http://lwn.net/Articles/79100/`

[5] `http://www.openmp.org`

[6] Nick Piggin, "MM patches," `http://www.kerneltrap.org/~npiggin/nickvm-267r1m1.gz`

[7] `http://www.spec.org/omp/results/ompl2001.html`

[8] `http://linux.bkbits.net:8080/linux-2.5/cset%4040213ca0d3eIznHTPAR_kLCsMZI9VQ?nav=index.html|ChangeSet@-1d`

[9] `http://www.cs.helsinki.fi/linux/linux-kernel/2002-01/1540.html`

Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

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