# Improving enterprise database performance on Intel Itanium® architecture

*Ken Chen, Rohit Seth, Hubert Nueckel*
Intel Corporation
*Software and Solutions Group*

## Abstract

In this paper, we will present several operating system features that improved database performance under OLTP[1] workload significantly, such as Huge TLB[2] page to reduce DTLB[3] misses as database uses large amount of shared memory and asynchronous I/O to accommodate high amount of random I/O without introducing the overhead of many I/O processes. We will also present many other kernel optimizations that were developed by Intel, Red Hat and the Linux community that improved the scalability and performance of Linux kernel, specifically the areas are: raw vary I/O, kernel data structure footprint reduction, global io_request_lock reduction, and storage device driver optimization.

## 1 Introduction

Linux has been receiving a great deal of attention in the past few years. The popularity is propelled by wide range of adoption of Linux for enterprise computing. Major software vendors have been supporting their products on Linux for many years. As the enterprise software solution stack builds up everyday, it is crucial that Linux kernel development takes this opportunity to ensure that kernel provides key necessary infrastructure for enterprise application to excel. This means developing enterprise focused operating system (OS) features, improving performance by extending the scalability, and many other areas.

Relational database management systems (RDBMS) are complex server applications that solve the problems of information management. The RDBMS reliably manages large amount of data in a multi-user environment such that users can concurrently access shared data. While it is required to maintain consistent data between users, it is also required to deliver high performance. All these requirements need high-quality infrastructure provided by the operating system. Some of the examples are virtual memory management for managing vast amount of physical memory, scalable I/O subsystem, robust / high performance storage subsystem, light-weight inter-process communication, and robust / high performance networking subsystem.

Recent Linux kernel development has addressed many of the areas with a focus to provide key functionality for enterprise workloads. The rest of the paper will discuss new kernel features as well as performance enhancements in the context of database running OLTP workload.

[1] On-Line Transaction Processing
[2] Translation Lookaside Buffer
[3] Data TLB

## 2 Overview

On-line transaction processing refers to a class of applications that facilitates and manages transaction-oriented operation, typically for data entry and retrieval transactions in a number of industries. The basic skeleton of OLTP environment consists of multi-tier software applications that allow thousands of users to concurrently execute transactions against a database. Typically transactions are either executed on-line or queued for deferred execution and have certain characteristics on the distribution between a mixture of different types. Because of the complexity and overall execution behavior of OLTP workload, the workload characteristics can be summarized as:

- Simultaneous execution of multiple types of transactions that span a breadth of complexity
- On-line and deferred transaction execution
- Significant random disk input/output
- Transaction integrity
- Unique distribution of data access
- Contention on data access and update

From system architecture perspective, the OLTP workload exercises a breadth of system components associated with the environment. Database server application and the underlying operating system software are the key software components to provide high performance. Earlier evaluation of Linux kernel under OLTP workload revealed several hot spots or limitations from performance point of view, such as large execution time spent in low level TLB miss handling, large number of process context switch due to blocking synchronous I/O, large execution time on functions related to I/O elevator algorithm, and large execution time spent on spinning on a highly contended lock like global io_request_lock. In the following sections, we will examine how features like Huge TLB and asynchronous I/O allow database application to exploit maximum hardware capability with minimum overhead from Linux kernel and how Linux I/O subsystem is improved to reduce kernel execution time.

## 3 Huge TLB Support in Linux

### 3.1 Motivation of Huge TLB page

A TLB (Translation Lookaside Buffer) is a hardware structure for virtual-to-physical address translations that supports high performance paged virtual memory system. Typically it is a scarce resource on a processor. Operating systems always try to make best use of the limited number of available TLB entries on a system. Orthogonally, with advancement of semiconductor technology that resulting in ever growing memory capacity, it becomes more and more feasible both technically and economically to populate tens of gigabytes of memory on a server. For example, HP server rx5670 can be populated with 48 GB of memory with 1GB DIMM[4], or even 96 GB with latest 2GB DIMM.

Database server applications generally use large amounts of system memory in order to efficiently manage the actual databases that are usually much larger than system memory. It typically utilizes shared memory segments among multiple database processes. The first area in shared memory segments, usually the largest, is the database buffer cache. It holds copies of data blocks read from datafiles. A data block is the smallest unit of storage space

[4] dual in-line memory module

managed by database server. The RDBMS actively manages the data blocks in the buffer cache. When a user process requires a particular piece of data, it searches through the buffer cache. If the data is already in the cache (a cache hit), it will read the data directly from memory. Otherwise data block will be copied from datafile on disk into memory (a cache miss). It is well known that accessing data from memory is several orders of magnitude faster than accessing data from disk. Therefore in production environment, system administrator will typically allocate as much memory as possible for shared memory segments in order to improve cache hit rate by maximizing buffer cache size. However, accessing large amount of memory combined with random data access pattern of OLTP workload, it puts lots of pressure on CPU's TLB resource. For example, assuming 16K page size for Linux-IA64, a 48 GB of process memory would need 3 million TLB translations. Or to look at from hardware point of view, an Itanium 2 processor's internal TLB resource would only cover 2 MB of virtual address space with 16 KB page size.

With vast amount of memory each application process access, there is a need to make each TLB mapping as large as possible to reduce TLB pressure. Large contiguous regions in a process address space, such as contiguous data, may be mapped by using small number of large pages rather than large number of small pages. It is also important to note here that OS kernel cannot blindly pick up a larger page size for all applications because it may cause lots of fragmentation and very poor utilization of large amount of physical address space. Thus a requirement for having a separate large page size facility from the operating system becomes more and more important in terms of functionality and performance.

### 3.2 Design and Implementation

To support large page size for user application to utilize processor's capability, Intel worked with the Linux community to introduce a new OS feature that exposes the hardware architecture for application to benefit from using huge page size without affecting many other aspects of the OS. This new feature is called Huge TLB page. Specifically the Huge TLB support is attempting to solve the following problems:

- Increase CPU TLB coverage / Reduce data TLB miss rate
- Reduce process's page table memory requirement
- Pin data pages in physical memory

The design goal of Huge TLB interface is to expose the hardware architecture to application. Mapping the kernel, or specialized devices such as frame buffers by using large mapping is a relatively straightforward exercise. It only affects very limited portions of the operating system code. However, virtual memory implementation in Linux kernel makes the basic assumption that there is only one page size for user applications. This one size is related to MMU page size supported by a specific architecture. For example, on IA-32 this page size is 4K, and on Itanium-based system, user page size is configurable at kernel build time to be either 4K, 8K, 16K or 64K. Itanium 2 processor actually provides concurrent multiple page size support (4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 256M, 1G and 4G). The current VM system is not suited for supporting multiple user page sizes because the knowledge of one page size is ingrained in several subsystems within the kernel. It is important to note that supporting multiple page sizes affects both architecture dependent and independent portions

of the Linux kernel. That is, a clean separation of architecture dependent and independent code in kernel is not enough to mitigate the difficulties of supporting multiple page sizes.

The allocation of Huge TLB page is performed in two phases. First a system administrator requests the kernel to reserve a set of memory in a special huge TLB page pool. The reservation of each huge TLB page is constrained that memory to be physically contiguous. Once huge TLB pages are reserved by the operating system, they can be used by application through two well defined system interfaces, either by mmap interface or through the standard System V shared memory interface. Note, application changes are required to use Huge TLB pages.

### 3.3 Application Benefit

To quantify the speed up of RDBMS under OLTP workload, we setup an experimental environment similar to industry standard OLTP benchmark on a Itanium 2 processor based platform.

First a baseline result is established with standard 16K page size. We then ran experiment with 256 MB page size while holding total memory in shared memroy segments constant. Throughput is then normalized to baseline. Figure 3.1 depicts the result.

We can easily see that with each incremental increase in page size used for data pages in shared memory segments, the speed up is notably at 11% overall for 256 MB huge TLB page size.

To further study how various page size speeds up the overall OLTP throughput at hardware micro-architecture level, we used Itanium-processor's hardware performance monitoring unit (PMU) to measure TLB pressure with various page size. The usage model of PMU


Figure 3.1: Relative OLTP throughput with various page size while holding database buffer cache size constant

are described in detail in several publications [1][2].

Again a baseline is established and data were collected for each page size. To measure hardware TLB pressure, we measured with metric of DTLB miss rate, or inverse of average number of data references per DTLB miss. As shown from figure 3.2, there is significant reduction in data TLB miss rate by using huge page size. For 256 MB page size, the DTLB miss rate is reduced by 65%, or inversely, the number of data references between successive TLB misses increases by 280%.


Figure 3.2: DTLB miss rate comparison

It is also interesting to observe that TLB pres-

sure for OLTP workload on Itanium 2 based system does not vary much with respect to total memory size, it is more or less a function of I/O load. For example, two experiments were conducted such that one with 16 GB database buffer cache while the other has 32 GB. The micro-architecture DTLB miss rate for both configurations are well within a couple of percentage points. This experiment points out that even at different OLTP throughput due to different size of database buffer cache, the benefit of using larger page size is equally significant. With 256 MB page size, the hardware TLB resource on Itanium 2 processor would be able to cover up to 32 GB of memory and primary source of TLB misses are shifted to data access to process's local data and task context switching.

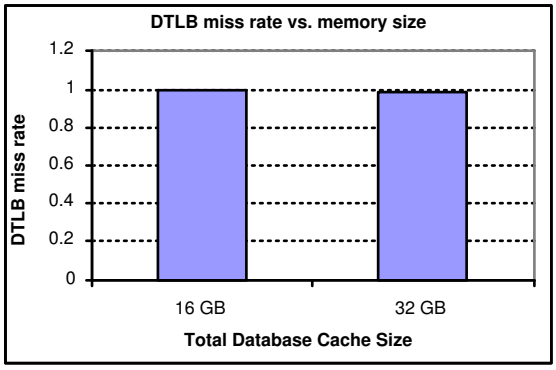


Figure 3.3: DTLB miss rate vs. memory size

A second benefit of using huge TLB feature is that the memory usage for process's page table is significantly reduced. Taking a 48 GB system as an example, if 45 GB is allocated as 180 256MB huge TLB pages, memory for page table covers that 45 GB of vma is only 1440 byte. For 100 processes that shares the 45 GB of shared memory segment, total memory for page table is 1.6 MB considering each process round up 1440 byte to one page. In the case of using normal 16 K page size, the memory requirement grows to 2250 MB (3 million entries * 8 bytes/entry * 100 processes, ignoring first and second level page table structure for simplicity). A secondary effect is that this 2.2 GB of memory reduced from page table can be better utilized by application to further increase application's performance.

A third benefit of huge TLB feature is that memory allocated for huge TLB page is pinned in physical memory and is not considered for swapping. This eliminates the chance of swapping physical pages that are being used for holding critical application data.

# 4 Linux I/O Subsystem

## 4.1 Dynamic vs. Static kiobuf allocation

Direct device access via raw devices partition improves database performance. A raw device partition is a contiguous region of a disk that can be accessed via a character device interface (/dev/raw on Linux). Such access typically bypasses the file system buffering. Since RDBMS does its own memory cache and I/O management, there is no need to have operating system to perform another level of caching and buffering. In fact, it is better to leave that task to application because it has much better information to determine optimal I/O strategy.

In a large OLTP workload configuration, due to sheer number of disk drives and the need to spread I/O load onto large number of disk drives, a database server typically opens large amount of data files where these data files reside on raw devices. Independent processes within the database server application will each open same set of data files.

The existing raw I/O code will statically allocate one kiobuf and its associated structures (mainly buffer_head structure, abbreviated as bh hereafter) upon every raw device open. There are 1024 bh allocated for each kiobuf. In a benchmark configuration, the memory re-

quirement just for the bh structure is calculated as following:

150 raw devices * 120 db processes * 1024 bh * 192 byte/bh = 3534 MB

However, since each process can have only one outstanding synchronous I/O at any given time, the active memory required for 120 processes are:

120 db processes * 1024 bh * 192 byte/bh = 24 MB

There are massive amount of memory being set aside by the bh structure and only 0.67% of them are being actively used. This large amount of under-utilized memory can be better devoted for other part of the system, for example, database buffer cache.

The cause of the issue is that each kiobuf structure is associated with a file descriptor. Although in certain cases, static bh allocation avoids the overhead of dynamic allocation, this static allocation scheme actually hurts performance for OLTP workload due to displacement of memory allocated for bh structure but otherwise can be used for database buffer cache.

To enable large number of raw devices to be opened simultaneously, we removed the static kiobuf allocation in raw_open function and at each invocation of rw_raw_dev function, kiobuf is dynamically allocated and freed for each raw I/O request. In order to reduce the prohibitive amount of overhead with dynamic allocation of all the memory arrays in kiobuf, we treat kiobuf and its associated member arrays as one entity. With the aid of constructor and destructor API provided by the kernel slab allocator, member arrays of kiobuf are allocated and initialized upon the creation of kiobuf object. Subsequent dynamic allocation would only incur one level of kmem_cache_alloc and kmem_cache_free overhead for such large data structure. With the per-CPU slab allocation area, the cost of dynamic allocation is even more affordable. The overhead of this dynamic kiobuf allocation is measured at 0.8 % for 2 KB I/O size and 0.1% for 128 KB I/O size.

It should be noted that even though the size of kiobuf structure is small (128 byte on Linux-IA64), the entire kiobuf entity is fairly large at 200KB. The per-CPU array for kiobuf slab cache should be managed pro-actively. With default parameter that calculates the per-CPU array size based on object size, there will be maximum 252 objects allocated on per-CPU array and on a 4 CPU system, this leads to 1008 kiobuf entity, or 200MB memory allocation. A small burden to the system administrator.

### 4.2 Variable size Block I/O

A second enhancement made to the raw device layer is to enhance the effectiveness for the raw vary I/O on Linux-IA64. The existing code restricts the sector combining to maximum size of RAWIO_BLOCKSIZE (4KB). The user pointer is also restricted to be aligned on that boundary (4K aligned). Both restrictions are sub optimal on Linux-IA64 because they reject many scenarios that can be put into speed path.

The implementation can be modified to be run time page size aware instead of hard coded constant value. The concept is to combine all sectors within a page to send down to submit_bh. For example, on a system with default page size of 16KB, a raw I/O request with 16KB size would be broken down to 4-4-4-4KB with existing code where it could be combined optimally as one 16KB request to submit_bh. The user pointer should only be restricted to sector aligned. For example, again on a system with page size of 16KB, a raw I/O request of 4 KB I/O size with user pointer

aligned on 2KB into a page would be rejected by the existing code for fast path consideration where technically it could be in the fast path. We measured the speed up varies from 10% to 280% with micro-benchmark depending on the I/O size and buffer alignment for this enhancement.

Again, using OLTP workload to measure how well the raw vary I/O and the enhancements measure up in production environment, we ran two experiments, one with and one without raw vary I/O. It was measured that raw vary I/O gives 4% performance advantage over one without for OLTP workload.

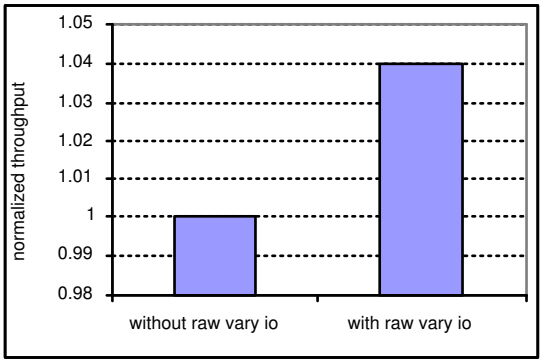


Figure 4.1: Comparison of raw vary I/O under OLTP workload

### 4.3 Relieve global lock contention

Another area of improvement in block I/O subsystem was the reduction of global io_request_lock usage. Much work has been done in this area [4] and the result of the work is incorporated in products released by several major Linux OS distributors. In earlier releases of Linux kernel 2.4, I/O requests are queued one at a time while holding the global lock io_request_lock. The Linux community has implemented many iterations/versions to break the global lock to per device lock. With the optimization, I/O requests are queued while holding a lock specific to the queue associated with the request. This improves concurrent I/O queuing and significantly improves I/O throughput.

## 5 Asynchronous I/O

### 5.1 History of AIO implementation

Several asynchronous I/O implementations following the POSIX standard were developed during the Linux kernel 2.3 development cycle. The implementations were either in kernel space or user space. Both of them employed an idea of an I/O queue with N number of helper threads that issues synchronous I/O to the underlying OS. However, there are several drawbacks with this approach for database application. First of all, even though the interface is asynchronous like, the I/O throughput is severely limited due to another layer of queuing. The optimum number of helper threads also depends on the characteristics of I/O subsystem and thus not flexible for wide range of production environment. A second issue is that the POSIX defined reap function aio_suspend() has a worst case of O(n) operation and tends to break down with large number of pending I/O.

### 5.2 A New paradigm

During the Linux 2.5 kernel development cycle, Red Hat kernel developers implemented a new AIO and its API based on the concept of completion queue [3]. Subsequently Intel worked with Red Hat to port and refined the AIO design for Linux kernel 2.4 on Linux-IA64.

The core of this kernel AIO implementation is centered around the completion queue. It introduces 5 new system calls for asynchronous operation. The core is generic that the operation is not just restricted to disk I/O, but also for network and file system I/O. The completion

queue is created by system call io_queue_init and destroyed via io_queue_release. New I/Os are submitted via io_submit and queued only if there is sufficient space in the completion queue to receive resulting event. When I/O is completed, a corresponding event is put into the complete queue and can be reaped via io_get_events. RDBMS application typically uses unbuffered I/O and combined with AIO infrastructure, disk I/Os are now being queued directly at block layer to exploit maximum concurrency for the capability of the underlying hardware devices.

### 5.3 AIO Evaluation and Optimization

We first turn our attention to evaluate how well does kernel asynchronous I/O performs under heavy disk I/O workload using micro-workload. The system under test has 3 fiber channel host adaptors connected to 180-disk Clariion towers. The disk towers are configured as 10 hardware RAID-0 disk drives and each RAID-0 drive has 10 raw partitions. A micro-benchmark program is then requesting AIO randomly on the 180 raw partitions with random offset (round to multiple of sector size). The I/O size is limited to 2KB and 16KB to limit the permutation of all other variables.

The micro benchmark basically throttles I/O to keep the system busy with at least N number of I/O pending at any given time. When number of pending I/O reduces to N, test program will batch next set of I/O with 'B' number of I/O in one AIO io_submit call. Completed I/O also gets reaped with each occurrence of AIO submit, i.e., program will reap approximately 'B' number of I/O in one io_get_events call.

The first experiment is to measure average CPU time spent on processing one I/O in the AIO request array. We sweep across the 'B' parameter from 32 to 1024 while holding N constant at 1000. The data was measured with pure CPU cycles spend on processing I/O excluding the wait time due to disk access latency. Figure 5.1 depicts the result.

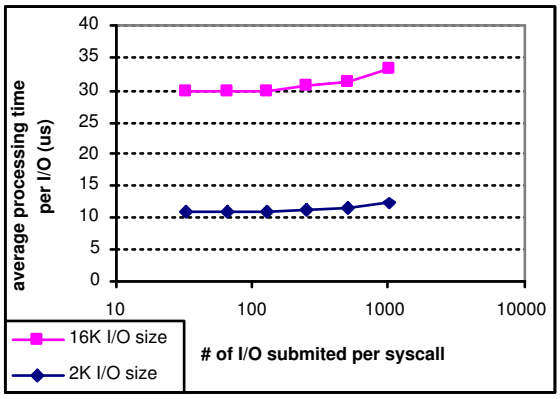


Figure 5.1: average AIO cost per I/O

For un-buffered I/O via raw device, the processing cost per AIO request in the most ideal case should be insensitive to the size of I/O. However, the large differences in the average cost between 2KB and 16 KB size in figure 4.1 indicate that there are some code path in the system break down badly with large I/O size. A kernel profiler showed that the elevator algorithm was responsible for the extra cost in the 16 KB case. It was apparent that raw vary I/O is also needed for asynchronous I/O path on raw device. Enhancements in addtion to raw vary I/O were also made in the generic AIO layer. Figure 5.2 illustrates the result of optimizations.

With raw vary I/O optimization, the cost of AIO on raw device is now quite consistent for different I/O size which matches to our expectation. The overall optimization improves 16 KB I/O size by 400% and 27% for 2 KB I/O size.

### 5.4 Application benefit

With all these fancy analysis done with micro-benchmark, the next question is how does

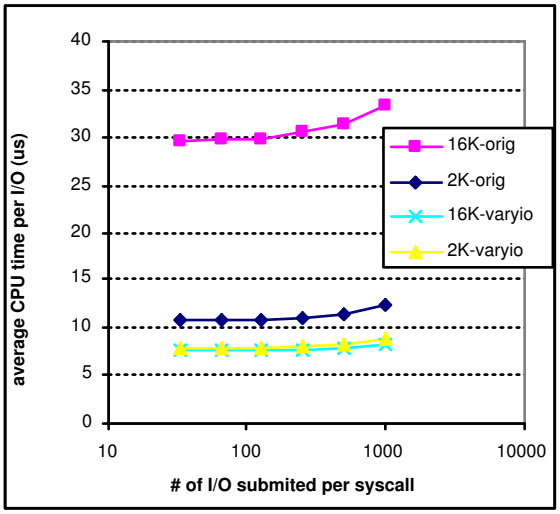


Figure 5.2: average AIO cost per I/O with optimization

AIO and the optimizations measure up in a real world production environment, like RDBMS with OLTP workload? Most I/O cache schemes employ deferred I/O operations and periodically sync up memory content with persistent data storage. A write back process is typically woken up on various conditions. One condition is database checkpoint where the process will write modified database records to persistent media in order to bring those copies of record in the persistent media current.

At high transaction rate and especially large percentage of update intensive queries in the OLTP transactions, the amount of modified database records existed in the buffer cache are high at the time when checkpointing initiates. It is essential that a write back process write those records to disks as quickly as possible to minimize amount of CPU processing time consumed on checkpoint task. Since the purpose of checkpoint is to sync-up persistent data file with content in memory, it actually has very little data dependency on when the blocks are being written, as long as database server gets notified that the writes are completed. This requirement fits perfectly with the non-blocking semantics of asynchronous I/O.

There are two ways for the writeback process to submit I/O to the OS. One is an internal RDBMS facility that distributes I/O among multiple helper processes for system that lacks the native AIO implementation. This facility is similar to I/O queue and help threads described earlier. The other is to submit I/O to the OS via native AIO interface. Again, two experiments were conducted, first with I/O helper threads configuration to establish a baseline result and second with native AIO configuration. Figure 5.3 depicts the result.

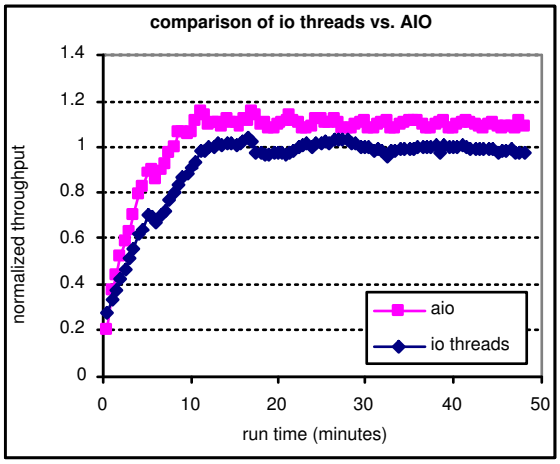


Figure 5.3: Benefit of AIO for OLTP workload

Several points worth noting here. At steady state, configuration with AIO is 10% higher in OLTP throughput compare to without AIO. It is due to combination of reduction in I/O processes' overhead and efficient OS I/O queuing and event reaping. Second note is that in the I/O helper thread configuration, system takes extra overhead in context switching between the helper threads and other active processes. Not only the helper thread takes a penalty hit with process context switch, it also puts more pressure on the CPU's data cache because more processes are actively running on the system. In the asynchronous I/O case, disk I/Os are

submitted directly to OS, thus reduces number of context switches. As illustrated from figure 5.4, the number of process context switches is reduced by 12 % with asynchronous I/O.

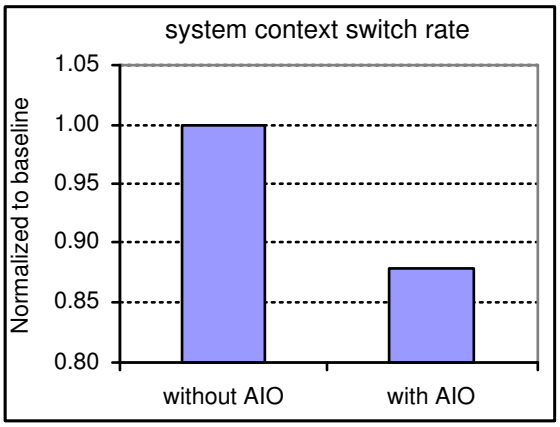


Figure 5.4: system context switch rate with and without AIO

There are many other secondary effects indirectly improving overall system performance by using AIO. System memory consumption is reduced because there aren't any I/O helper threads at all. OS scheduler will have less pressure because less number of active tasks it need to manage, and lastly inter-process communication overhead between the helper threads are eliminated. All of these translate into highly efficient scalable asynchronous I/O layer and higher OLTP throughput.

## 6 Storage Device Driver Optimization

While most I/O enhancements outlined in previous section are more or less transparent to storage device driver, some still do require cooperation from each individual driver to enable specific optimization. One example would be HP's smart array family of disk controllers. Since this driver hooks directly into Linux I/O block layer, it missed out all the enabling infrastructures for the raw vary I/O and the global io_request_lock optimization implemented for SCSI devices.

Both optimizations are fairly straightforward to enable. What we did was at the time of the controller's initialization, we initialize a per-controller raw vary I/O capability array and then hook that array into the blkdev_varyio defined in the block layer. To enable per device request lock, two locks are added in the controller's data structure, one lock for I/O request queue, and one for the controller itself. Locking primitives are then modified to use the corresponding request queue lock in the case of I/O queuing/dequeuing. For operations that pertain to controller, the controller lock will be used.

Other optimizations that were also actively worked on for this particular storage device driver are interrupt coalescing, 32-bit DMA command pool, and Itanium architecture specific command structure alignment.

## 7 Conclusions

In this paper we have outlined some of the key operating system requirements for running a high performance database on Linux. Implementations of huge TLB support and asynchronous I/O have been described along with how these features perform to expectation under OLTP workloads. The I/O subsystem for the Linux kernel 2.4 has been improved significantly to achieve high concurrency and efficiency for high demand I/O workload. Storage device driver optimizations are also shown to be equally important to materialize optimizations done at generic layer.

## 8 Acknowledgement

The authors of this paper would like to thank the following people who enthusiastically con-

tribute directly or indirectly to this paper, in no particular order: Asit Mallick, Arun Sharma, Tony Luck, Sunil Saxena, Mark Gross and several other groups at Intel Corporation; Red Hat kernel developers; Linux community around the world.

## References

[1] *Intel Itanium Architecture Software Developer's Manual*, Volume 1-3.

[2] David Mosberger and Stephane Eranian, *ia-64 linux kernel design and implementation.*, Prentice Hall, 1st edition, 2002.

[3] Benjamin LaHaise, *An AIO Implementation and its Behavior*, Ottawa Linux Symposium proceedings 2002.

[4] Peter Wai Yee Wong, et al., *Improving Linux Block I/O for Enterprise Workloads*, Ottawa Linux Symposium proceedings 2002.

#### Trademarks

# 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.*