Optimizing Operating System Performance for CC-NUMA Architecture

Moon-Seok Chang

IBM, MS 905-7C020, 11400 Burnet Rd., Austin, TX 78758, USA

Correspondence to:
Moon-Seok Chang

IBM
MS 905-7C020
11400 Burnet Rd.
Austin, TX 78758
USA

E-mail: cmoon@us.ibm.com

Tel: (512)-838-8473
Fax: (512)-823-8487

Sponsor:
ETRI(Electronic and Telecommunication Research Institute)
SCO(Santa Cruz Operation Inc.)

Optimizing Operating System Performance for CC-NUMA Architecture

Summary

CC-NUMA (Cache-Coherent Non-Uniform Memory Access) architecture is an attractive solution to scalable servers. The performance of a CC-NUMA system heavily depends on the number of accesses to remote memory through an interconnection network. To reduce the number of remote accesses, an operating system needs to exploit the potential locality of the architecture. This paper describes design and implementation of a UNIX-based operating system supporting CC-NUMA architecture. The operating system implements various enhancements by revising kernel algorithms and data structure. This paper also analyzes the performance of the enhanced operating system by running commercial benchmarks on a real CC-NUMA system. The performance analysis shows that the operating system can achieve better performance and scalability for CC-NUMA by kernel data striping, localization, and load balancing.

***Keywords**: CC-NUMA, UNIX operating system, remote memory access, data striping, localization, load balancing*

1. Introduction

CC-NUMA (Cache-Coherent Non-Uniform Memory Access) architecture is an attractive solution to scalable servers. It offers better scalability than traditional SMPs (Shared-Memory Multiprocessors) where all the processors share the common bus. The CC-NUMA architecture overcomes the scalability limits by distributing physical memory into multiple local memory modules, while providing a single system image and global common address space shared by all processors [1][2].

A CC-NUMA system consists of multiple nodes connected by an interconnection network. Since a processor accesses remote memory through the network, the remote access latency is much larger than the local access latency. In addition, an access to remote memory increases the network traffic for maintaining cache coherency, which significantly degrades the system performance. Such architecture poses many challenges in operating system design, implementation, and performance optimization.

In this paper, we present design and implementation of a CC-NUMA operating system based on SCO UnixWare 7[1] [3]. Since UnixWare 7 was originally designed for SMPs, we revised the global data structure and algorithms to exploit the potential locality of CC-NUMA architecture. We also implemented numerous kernel enhancements to maximize the performance. The major enhancements include large physical memory management, process scheduling, data localization, multi-path I/O, load balancing, and NUMA-specific APIs.

In the following sections, we will describe the kernel enhancements and their contribution to the overall performance. Then we evaluate the operating system performance by running SPEC SDET and AIM VII SS commercial benchmarks on a real CC-NUMA system.

2. Related work

A CC-NUMA system consists of multiple nodes or CPU groups (CGs) connected by a high-speed interconnection network. Each node contains a set of processors, local memory, and a custom node controller. The node controller is responsible for maintaining the consistency of data distributed into multiple local memory modules belonging to each node. Figure 1 illustrates an overall architecture of a typical CC-NUMA system.

In contrast to a SMP machine, a CC-NUMA system has different access times for different parts of memory. When requested data are missing in the processor cache and local memory, they are retrieved from a remote memory residing on another node. Since the remote data are accessed through an interconnection network, the remote access latency of a typical CC-NUMA system is at least two or three times the local access latency even without memory contention [2][4].

To reduce the amount of traffic in the network, the node controller may contain FMC (Far Memory Cache) storing the remote data temporarily. But if the requested data miss the FMC, the processor has to wait until

[1] UnixWare 7 is a trademark of SCO (Santa Cruz Operation Inc.).

the data are served from the remote memory through the network. Such large latency in the remote memory access makes the localization of the memory access the most important performance issue.

There have been many researches on CC-NNUMA architecture in academy, including Stanford DASH [2] and DASH [5], MIT Alewife [6], and University of Toronto NUMAchine [7]. But most of these researches focused on CC-NUMA hardware performance, and only few of them focused on the operating system support and its performance.

As a part of research on Stanford DASH, Rohit et al. evaluated the effect of operating system scheduling and page migration policies on the CC-NUMA performance [8]. They performed experiments using sequential and parallel applications as a workload. With given workload, they claimed that an operating system policy incorporating cache affinity and page migration could double the performance.

Verghese et al. studied on dynamic page migration and replication [4]. They modeled a CC-NUMA machine on the SimOS simulation environment. With the simulation results, they claimed that dynamic page migration and replication substantially increase the application performance and reduce the contention in the CC-NUMA memory system.

Chapin et al. characterized the performance of SGI IRIX, a variant of UNIX SVR4 operating system [9][10]. They analyzed the memory performance of the operating system using a cache miss monitor. Through the experiment, he proposed several enhancements for operating system such as kernel code replication and memory hotspot reduction. They claimed that SMP-based operating systems could be modified to achieve reasonable performance on CC-NUMA architecture without significant changes in the kernel data structure and algorithm.

A research group at University of Toronto proposed Tornado operating system specially designed for CC-NUMA architecture [11]. This group introduced new design concepts supporting CC-NUMA such as clustered objects, a protected procedure call facility, and an encapsulated locking strategy. They claimed that they could improve the performance by redesigning the operating system from the scratch.

The academic researches and prototypes have been followed by commercialization of several CC-NUMA systems. Examples are Sequent NUMA-Q, DG Aviion 20000, and SGI Origin.

Sequent NUMA-Q is targeted toward commercial workloads such as database and transaction processing [12]. NUMA-Q consists of homogeneous processing nodes connected by a high-speed ring interconnection. Each processing node is an Intel Quad bus-based SMP with four Pentium Pro processors. The customized interconnection board implements the SCI (Scalable Coherence Interface) directory protocol.

DG Aviion 20000 also uses Pentium Pro Quads as their nodes [13]. The multiple nodes are connected by SCI compliant Dolphin interconnection network that links those nodes in a bi-directional ring. Each node has FMC (Far Memory Cache) that stores remote memory references in order to reduce the amount of traffic across the link. Aviion 20000 runs DG/UX or SCO UnixWare 7 operating system optimized for its architecture.

Recently IBM built a CC-NUMA prototype hardware running Windows NT operating system [14]. This prototype also uses Intel Quad SMP as building blocks and implemented programmable performance monitors to measure the frequency of remote memory accesses. However, They just ported operating system to their prototype without modification or optimization in kernel data structure and algorithm.

This study presents performance optimization and evaluation of UnixWare 7 operating system for CC-NUMA architecture. While previous performance studies are based on scientific workloads on prototype machine or simulation environment, this study evaluate the performance of a real CC-NUMA system running SDET and AIM VII commercial workloads. In addition, this study focuses on performance optimization of the operating system, not hardware platform. In the next section, we will describe the operating system support and kernel enhancements for CC-NUMA architecture implemented in UnixWare 7 in detail.

3. Operating system support for CC-NUMA architecture

3.1 Kernel enhancements

SCO UnixWare 7 is one of latest UNIX operating systems runs on Intel processor platforms ranging from small-scale SMPs to enterprise class multiprocessors. UnixWare 7 implements various kernel enhancements to for CC-NUMA architecture. In this section, we describe the major enhancements such as kernel code replication, kernel data striping, data localization, multi-path I/O, virtual memory management, load balancing and migration, and CC-NUMA specific APIs.

(1) Kernel code replication

The kernel code and data are most frequently accessed area in the system. A previous study reported that a large fraction of execution time is spent on accessing the kernel code and data [9]. UnixWare 7 reduces the number of remote accesses by simply replicating the kernel code to each node. At boot time, the kernel replicates its code to local memory on each node and maps its virtual address space to the local copy.

Generally, the kernel code replication does not cause the consistency problem, because it is read-only accessible in most of cases. However, any changes to kernel code that has taken place after replication must be propagated to each node. For example, when a kernel debugger plants a software breakpoint, it has to modify every copy of affected instructions replicated to each node.

(2) Kernel data striping

The kernel data area are classified as two groups: read-only and write accessible. The read-only kernel data can be replicated without consistency control. For example, kernel page tables, the page slice table, kernel virtual address map, scheduler dispatch tables are read-only accessible and can be replicated to each node.
On the contrary, most of kernel data are write accessible and cannot be replicated easily. Those kernel data are stripes across the nodes with a granularity of one page. The kernel data striping eliminates the memory hot spots by distributing memory accesses to multiple nodes.

(3) Kernel data localization

To increase the locality, the kernel global data structure has been revised to have per-node structure. For example, the system-wide global run queue has been replaced by per-node run queue, and various metrics are collected on a node basis.
In addition, the kernel maintains a new data structure called *cglocal* to keep all the variables local to a node. Other global data structure such as page lists, anon slots, KMA(Kernel Memory Allocator) pools, and memory reservations are also maintained on a per-node basis. Each node has its own instance of kernel daemons such as *fsflush_percg()*.

(4) MPIO

MPIO (Multi-Path I/O) supports multiple I/O channels to access the same storage unit. It enables an operating system to make multiple decisions in routing an I/O to balance the load or bypass non-functional paths. MPIO increases the availability by re-routing the I/O request, and improves the I/O throughput by allowing the kernel to choose the nearest path.

To maximize the I/O performance, CC-NUMA I/O hardware needs to be fully connected, that is, the system needs to have a near disk controller available on each node. The MPIO improves significantly the I/O performance of a CC-NUMA system by distributing I/O requests across the nodes and eliminating the I/O imbalance problem.

To improve the network performance, STREAM subsystem has been redesigned to have per-node data structure. An instance of network driver is bound to each node and scheduling algorithms are implemented per-node basis.

(5) Virtual memory management

The virtual memory design also has been revised to support large physical memory in CC-NUMA architecture. A number of virtual memory interfaces have been changed for localization purposes, and a number of new interfaces have been implemented.

In page pool management, each node has its own physical page pool. When a page is requested on a node, it comes from the local page pool by default, and return back to the local pool when it is freed. Since the page pools are local to each node, paging daemons such as *pageout* and *fsflush* page in and out local to each node and they handle only local page lists.

The *vnode* management provides several placement policies specific to CC-NUMA architecture. A *vnode* page can be bound to a specific node, or replicated to each node if it is a read-only file or text file.

Kernel memory allocator (KMA) has been changed to have preference to allocate memory local to the calling processor. KMA keeps per-node free lists and per-CPU free lists. A kernel memory can be allocated from per-CPU free lists only if a local CPU belongs to a node for which allocation request was made. Per-node free lists contain memory only from the local node. KMA also provides a new interface to allow a caller to allocate kernel memory in the indicated node.

(6) Load balancing and migration

Load balancing distributes the workload equally among the nodes. We implemented two load balancing policies: static and dynamic. Static load balancing is performed only at specific times during the lifetime of a process, either before the process begins (*fork(2)* time) or when the process instantiates a new address space (*exec()* time). The kernel identifies the most lightly-loaded node and creates a process on such node using resource metrics on each node such as *vmmeter()*. The *vmmeter()* maintains the resource usage information such as the amount of available memory and the length of the ready queues.

In contrast, dynamic load balancing periodically checks the load among the nodes. If the kernel detects that the imbalance exceeds a specified limit, a process migrates from the most heavily loaded node to the mostly lightly loaded node. We implemented the placement policy based on three criteria: run queue length, processor idle time, and the amount of free memory.

A process can migrate to a different node for a very brief period only with the minimal memory structures required to execute, and return to its original node. Otherwise, the kernel may change the home node and move all memory objects associated with the process including kernel data. The dynamic load balancing and migration policy are useful for maximizing the system utilization, but it may not improve the throughput in case of excessive migration.

3.2 NUMA-specific APIs

UnixWare 7 provides new APIs to support CC-NUMA architecture. These APIs are provided to meet the needs of the applications targeting the CC-NUMA system. The APIs are implemented in a dynamically-linked shared library. The interfaces and their functions are listed below. Note that CG(CPU Group) and node are used interchangeably in the rest of this paper.

int **cg_ids**(int *selector*, int *ncgids*, cgid_t **array*)

Retrieve a list of the CGs selected by *selector*

int **cg_processors**(cgid_t *cgid*, int *selector*, int *nprids*, processorid_t **array*)

Retrieve a list of the processors belonging to CG *cgid* and selected by *selector*

int **cg_info**(cgid_t *cgid*, cginfo_t **infop*)

Retrieve information about the specified CG.

int **cg_bind**(idtype_t *type*, id_t *id*, cgid_t *cg*, int *flags*, cgid_t **ocg*, int **oflags*)

Bind or query the binding of the specified process or LWP.

cgid_t **cg_current**(idtype_t *type*, id_t *id*)

Return the identity of the CG to which the specified process or LWP belongs

int **cg_memloc**(caddr_t *addr*, size_t *len*, cgid_t **vec*)

Determine which CG instantiates the specified pages of the caller's address space

4. Experiment

4.1 Experimental environment

In the previous section, we described the UnixWare 7 kernel enhancements for CC-NUMA architecture. In this section, we evaluate the performance contribution of those enhancements by running SPEC SDET and AIM VII SS [15] commercial workloads.

The SPEC SDET benchmarks simulate multi-user workloads in a software development environment. SDET benchmark runs numbers of concurrent processes invoking typical UNIX commands such as make, cp, rm, nroff, grep, etc. The throughput is measured in scripts per minute as a function of the number of simulated users. The AIM VII SS (Shared System Mix) benchmark also measures the throughput in AIM jobs per minute versus simulated application load tasks. The SDET and AIM VII SS benchmarks are widely used for measuring the operating system performance, since both benchmarks spend over 50% of total execution time in the kernel mode.

The hardware platform used in this experiment is a Data General Aviion 20000. Each node contains four 200 MHz Pentium Pro processors with 512 KB L2 cache and 1.5 GB main memory. An SCI compliant Dolphin interconnection links the nodes in a bi-directional ring with bandwidth of 2 x 400 MByte/sec. Each node has 32 MB FMC (Far Memory Cache) that stores remote memory references temporarily.

To meet the I/O bandwidth requirement of the benchmark, we configure the platform system with DG Clariion RAID disks. We bind 10 disks for a RAID-0 logical disk, and assign four logical disks and two Qlogic 1040 SCSI controller to each node of the platform system. For each run, the benchmark program is instrumented by *sar(1M)* and kernel profile data are collected by *profiler(1M)* utility.

4.2 CC-NUMA vs. SMP_ON_CCNUMA

To evaluate the contribution of the kernel enhancements for CC-NUMA, we first compare the performance of two versions of a kernel: CC-NUMA and SMP_ON_CCNUMA.

The SMP_ON_CCNUMA version enables an SMP kernel to run on a CC-NUMA machine, making use of all of the processors, memory, and I/O devices in the system. This version is developed to measure the comparative performance of a CC-NUMA kernel relative to an SMP kernel on the same hardware. In this version, the operating system considers a CC-NUMA machine as an SMP machine having only one node, which turns off all the kernel enhancements for CC-NUMA architecture.

The CC-NUMA version is a base kernel that enables all of basic enhancements for CC-NUMA, including kernel code replication, kernel data striping, kernel data localization, and MPIO, except for load balancing and migration. The performance effect of load balancing and NUMA-specific APIs will be evaluated in the next subsection.

The SDET benchmark requests operating system services intensively, spending over 60 % of execution time in the kernel. The measurement result shows that the throughput increases initially with the workload and reaches a peak value. After the peak value, the throughput levels off, then falls off as the workload is further increased.

Figure 2 illustrates the throughput of CC-NUMA and SMP_ON_CCNUMA versions in a single node configuration with four processors. Figure 2 shows that the SMP_ON_CCNUMA version performs little better, because the CC-NUMA version has slight overhead due to the kernel modification. But the overhead is 4.7% at maximum, and becoming negligible as the number of users increases.

Figure 3 illustrates the improved throughput of the CC-NUMA version relative to SMP_ON_CCNUMA in a 2-node configuration. In Figure 3, the throughput of the CC-NUMA version exceeds that of SMP_ON_CCNUMA by 10 – 28%. For example, the CC-NUMA version scores the peak throughput of 5142.9 scripts/min at 24 simulated users, which is 13.2% higher than SMP_ON_CCNUMA. At 512 users where the system is saturated, the throughput drops to 3445.2 scripts/min, but still 27.8% higher than SMP_ON_CCNUMA.

Note that the results in Figure 2 and 3 do not show the optimal performance, because CC-NUMA version does not enable load balancing and dynamic migration, and the SDET benchmark does not use NUMA-specific APIs.

The impact of the CC-NUMA enhancements on the kernel behavior can be found in the kernel profile data collected by *profiler(1M)*. Figure 4 illustrates the three most frequently used kernel routines in two versions at 24 users in 2-node configuration. In the SMP_ON_CCNUMA version, the processors in the system spend 15.1% of time in *lock()* routine and 7.8% in *bcopy()*. In the CC-NUMA version, these values are reduced to 8.8% and 4.9%, respectively. This result indicates that the kernel enhancements, such as kernel data localization, reduce a large amount of lock contention in the CC-NUMA architecture.

Figure 5 illustrates the scalability of the CC-NUMA version with the increasing number of nodes. When comparing the peak throughputs, the CC-NUMA version increases the performance by 19% for a 2-node configuration relative to a single node configuration. In terms of node scalability, the CC-NUMA version scales only 60% (=119% / 2 nodes), which is higher than SMP_ON_CCNUMA, but lower than expected.

The reason for the poor scalability of the CC-NUMA version can be explained by the system activity report shown in Table 1. Table 1 shows the percent of time waiting for I/O (%wio) and idling (%idle) reported by *sar(1M)* in a 2-node configuration. In the SMP_ON_CCNUMA version, the %wio and %idle continuously decrease to less than 1% as the number of users increases. In the CC-NUMA version, however, the %wio and %idle do not decrease continuously, but remain much higher. Especially %idle is 7% at minimum, and remains as high as 12 - 38%.

The higher percentages in %wio and %idle imply that the workload is unbalanced and I/O requests are not uniformly distributed over the nodes. The load imbalance problem results in poor scalability worse than expected in the CC-NUMA version. This result suggests that the operating system needs to provide a load balancing facility to fully utilize the resource in the CC-NUMA system.

No. of Users		1	8	16	24	32	64	128	256
SMP_ON_ CCNUMA	% wio	43	30	12	5	4	2	2	1
	% idle	49	4	4	3	3	2	0	0
CC-NUMA	% wio	24	28	14	6	9	4	9	8
	% idle	70	15	7	15	12	38	29	25

Table 1. Average %wio and %idle for SDET (2-node configuration)

4.3 Load balancing

The experiment results in the previous subsection showed that a CC-NUMA operating system may suffer from poor scalability due to the load imbalance problem. To solve this problem, we implemented three load balancing policies: static load balancing(SLB), dynamic load balancing(DLB), and binding.

The two load balancing policy can be enabled exclusively. When static load balancing is enabled, the operating system creates a process on the most lightly loaded node at *fork()/exec()* time. When dynamic load balancing is enabled, the kernel periodically checks the load among the nodes and migrates a process from the most heavily loaded node to the most lightly loaded node. The DLB policy also includes load balancing on process creation time. Both SLB and DLB policy uses a simple resource usage metric on each node based on the length of the run queues, processor idle time, and the amount of available memory.

In Binding policy, a process can be bound explicitly to a specific node during its entire life without migration. We also implemented a set of NUMA-specific APIs supporting process binding: *cg_info()* and *cg_bind()*. These APIs allow a user to bind explicitly a process to a specific node to distribute the load across the nodes.

In this subsection, we evaluate the performance improvement for each policy by running AIM VII SS benchmark. The AIM VII SS benchmark bas been simply modified to use NUMA-specific APIs for binding. The experiment is performed on 3-node configuration consisting of 12 CPUs and 4.5 GB memory in total.

Table 2 shows the AIM VII SS throughput of CC-NUMA kernels implementing three different load balancing policies. Similar to SDET throughput, the AIM VII SS throughput initially increases with simulated loads and reaches a peak performance value before leveling off, or in some cases falling off as the load is further increased. In DLB policy, the load is checked every second, and the process migration is set to activate when the difference between per-node run queue lengths becomes larger than two processes.

In Table 2, the DLB improves the performance by 7.6 – 8.4% at 80 and 96 simulated loads in comparison with SLB. The experiment also shows that Binding policy performs far better than DLB as well as SLB. For example, Binding policy scores the peak performance of 2763.6 jobs/min at 96 simulated loads, which is 13.0% and 17.6% higher than DLB and SLB policies.

No. of Users	32	48	64	80	96	128	256
SLB	1803	2270	2396	2256	2272	2277	2207
DLB	1755	2331	2469	2445	2444	2393	2228
Binding	1930	2388	2518	2622	2763	2532	2292

Table 2. AIM VII SS throughput for SLB, DLB, and Binding policies on 3-node configuration

For further investigation, we analyze %wio and %idle for each policy in Table 3 and 4. Table 3 shows average %wio on each node. For example, at 32 simulated loads, DLB policy spends 3%, 2%, 9% of total execution time in waiting for I/O on node 0, node 1, node 2, respectively.

In Table 3, %wio has been reduced to less than 3% on reasonable loads for all of three policies. The small percentages in %wio imply that I/O requests have been distributed evenly across the nodes. This result also indicates that the platform system provides enough I/O bandwidth, eliminating the effect of I/O from the cause of performance differences between the load balancing policies.

Table 4 illustrates the average %idle on each node, which identifies the cause of the performance difference between three policies. In Table 4, DLB policy performs better than SLB in reducing %idle, but its reduction rate is less than expected. A further reduction of %idle can be achieved by using Binding policy. At peak performance, Binding policy reduces maximum %idle to 12% at 96 simulated loads, while SLB and DLB have 53% and 43% of maximum %idle.

Table 4 shows that DLB policy performs better than SLB, but is not sufficient to maximize the performance of the CC-NUMA system. The inefficiency of DLB policy comes from the limited accuracy in the resource metric. The resource metric needs to be updated as frequently as possible to provide accurate information for the processor and memory usage. The dynamic nature in memory allocation and deallocation makes status information less accurate, and also increases the overhead in the collection activities. In addition, DLB may not be useful if the process lifetime is not long enough to gain the performance benefit from the migration.

Table 4 suggests that Binding policy can maximize the performance once the workload is well known and distributed evenly across the nodes. Binding policy can be easily implemented by using *cg_ids()* and *cg_bind()* NUMA-specific APIs as described in Section 3.2

No. of Users	32	48	64	80	96	128	256
SLB	5,2,2	8,1,2	4,1,1	3,1,1	4,0,0	4,1,1	2,1,1
DLB	3,2,9	2,1,5	1,1,4	1,1,3	1,1,3	0,0,1	0,0,1
Binding	3,3,2	1,4,4	1,1,1	0,0,0	0,1,1	0,0,0	0,0,1

Table 3. Average %wio for AIM VII SS on 3-node configuration

No. of Users	32	64	96	128	256
SLB	53,28,28	50, 6, 6	50, 7,10	50, 7, 6	50, 5, 5
DLB	35,30,86	13, 9,41	20, 7,40	16, 3,30	13, 3,33
Binding	35,47,37	17,22,22	10,12,12	8,13,11	3, 7, 7

Table 4. Average %idle for AIM VII SS on 3-node configuration

4.4 Scalability

In this section, we perform scalability test by running AIM VII SS on the platform with the increasing number of nodes. To maximize the performance for each configuration, we enable Binding policy as well as all of kernel enhancements described in the previous sections.

Figure 6 illustrates the node scalability, the peak throughput of AIM VII SS for increasing number of nodes. In Figure 6, the peak performance increases as the number of node increases: 1505.9 jobs/min on a single node, 2377.8 jobs/min on 2-node, and 2763.6 jobs/min on 3-node configuration. The normalized peak performance values are 1: 1.58: 1.84. That is, the node scalability of the platform system is 79% with 2-node, but drops to 61% with 3-node configuration. This result indicates that the platform system offers reasonable scalability with up to two nodes, but still limited scalability beyond three nodes.

4.5 Remote memory access

To identify the primary factor for the poor scalability beyond 3-node configuration, we first measured the number of remote accesses during the benchmark run. The platform system contains FMC (Far Memory Cache) for each node to reduce the number of remote accesses as well as performance counters for monitoring the FMC misses. The FMC in the platform system has a 32 MB DRAM cache with 4-way associativity and 64-byte block size.

Table 5 shows the number of FMC misses per instructions executed on each node during the AIM VII SS benchmark run. In the case of AIM VII SS, the FMC miss rates are extremely low. The FMC miss rate would be dependent on the workload, but many of commercial workloads are expected to have similar access patterns, because AIM VII SS is an operating system intensive benchmark invoking typical UNIX commands.

Table 5 suggests that remote access is not the primary factor for the poor scalability beyond 3-node configuration. This result also implies that the number of remote accesses has been reduced dramatically by various kernel enhancements for CC-NUMA.

No. of Users	32	48	64	80	96	128	256
Executed instr. (x 10^7)	181	268	347	532	650	985	2333
FMC miss rate per 10^7 instr.	1.96	1.22	1.12	0.90	0.87	0.78	0.73

Table 5. FMC misses for AIM VII SS on 3-node configuration

4.6 Spin lock contention

Next, we measured the time spent on spin lock contention. A spin lock contention refers to the situation in which two or more processors try to acquire the same lock simultaneously. This contention consumes lots of processor cycles since a processor executes in a tight loop, trying to acquire a spin lock which is held by another processor. A spin lock contention also consumes bus cycles in the form of cache coherence traffic. In addition, it may also cause extra cache coherence traffic arising from false sharing. The false sharing problem may cause more serious performance problem on CC-NUMA system when the spin lock variables are allocated in the same cache line of the FMC [1].

The processor cycles consumed by spin lock contention can be measured by kernel profiling tool facility. Figure 7 shows the percent of time spent in kernel spin locks instrumented by *profil(1M)*. Using kernel profiling, we found that kernel spin locks consume at peak performance (96 users) 12.25% of processor cycles on 3-node configuration, while 4.24% and 2.97% on 2-node and 1-node configuration, respectively. At 256 simulated users with 3-node configuration, the kernel spin locks consume as much as 23.78% of total execution time. This result indicates that the spin lock contention is expected to increase exponentially as the number of nodes increases, resulting in a severe scalability problem of a CC-NUMA system.

Through the individual lock instrumentation, we found that two major locks, *vm_pagefreelock* and *dnlc* lock are responsible for the most of time spent in spin lock contention. In UnixWare 7 implementation, *vm_pagefreelock* is a single global spin lock protecting the page free lists, and *dnlc* lock is a single global spin lock protecting *directory name lookup cache*. The contention on *vm_pagefreelock* is expected to increase as the size of physical memory increases, becoming more serious on a CC-NUMA system supporting large physical memory exceeding 4 GB. This result suggests that these spin locks should be split into finer granularity locks to improve the scalability of CC-NUMA system beyond 3-node configuration.

5. Conclusion

The CC-NUMA architecture demands operating system support for performance optimization. In this paper, we presented performance optimization of an UNIX-based operating system for CC-NUMA architecture. We implemented numerous kernel enhancements for CC-NUMA architecture, including kernel code replication, kernel data striping, virtual memory management, kernel data localization, multi-path I/O, load balancing and migration, and NUMA-specific APIs. This paper also evaluated the operating system performance using SDET and AIM VII commercial benchmarks.

To evaluate the performance contribution of the kernel enhancements, we first compared the performance of CC-NUMA and SMP_ON_CCNUMA versions. Through the experiment, we found that the CC-NUMA version improves the performance by 27.7% at maximum in comparison with the SMP_ON_CCNUMA. We also found that kernel enhancement in CC-NUMA version reduced the processor cycles spent in lock contention and memory copy to the half those of SMP_ON_CCNUMA.

We also evaluated the efficiency of three load balancing policy. The experiment results showed that Binding policy performs better than DLB as well as SLB by reducing the processor idle times. The results suggested that NUMA-specific APIs can play an important role in scheduling and migration.

Finally in the scalability study, the platform system showed 79% of scalability with 2-node configurations and 61% with 3-node configurations. The poor scalability beyond 3-node configuration is due to the spin lock contention problem rather than remote accesses. This implies that the kernel enhancements presented in this paper are very useful in reducing the number of remote accesses, making the CC-NUMA technology more promising.

References

[1] David Culler, Jaswinder Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1998.

[2] D. E. Lenoski and W. Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers. San Francisco, 1995.

[3] Rohit Chawla and Steve Baumel. Managing more physical with less virtual. May 2000; 30(6):639-661.

[4] Ben Verghese et al. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VII*. ACM, 1996; pp. 279-289.

[5] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. *Proc. of 21st International Symposium on Computer Architecture (ISCA)*. ACM, 1994; pp. 302-313.

[6] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. *Proc. of 22nd International Symposium on Computer Architecture (ISCA)*. ACM, 1995; pp. 2-13.

[7] A. Grbic et al., Design and Implementation of the NUMAchine Multiprocessor. *Proc. of 35th Design Automation Conference*. IEEE, June 1998. pp.66-69.

[8] Rohit Chandra et al. Scheduling and Page Migration for Multiprocessors Compute Servers. *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS) VI*, ACM, 1994.

[9] John Chapin et al. Memory System Performance of UNIX on CC-NUMA Multiprocessors. *Proc. of SIGMETRICS*. ACM, 1995; pp. 1-13.

[10] James Lauden and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. of 24th International Symposium on Computer Architecture (ISCA).* ACM, 1997; pp. 241-251.

[11] Ben Gamsa et al. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. *Proc. of Operaing System Design and Implementation (OSDI) III* , USENIX, 1999; pp. 87-100.

[12] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. *Proc. of 23rd International Symposium on Computer Architecture (ISCA).* ACM, 1996; pp. 308-317.

[13] R. Clark and K Alnes. An SCI Chipset and Adapter. *Proc. of Hot Interconnect V*. IEEE, 1996.

[14] B. C. Brock et al. Experience with building a commodity Intel-based ccNUMA system. *Journal of Research and Development,* IBM, March 2001; 45(2):207-228.

[15] Ran Giladi and Niv Ahituv. SPEC as a Performance Evaluation Measure. *IEEE COMPUTER*. Aug. 1995; pp. 33-42.

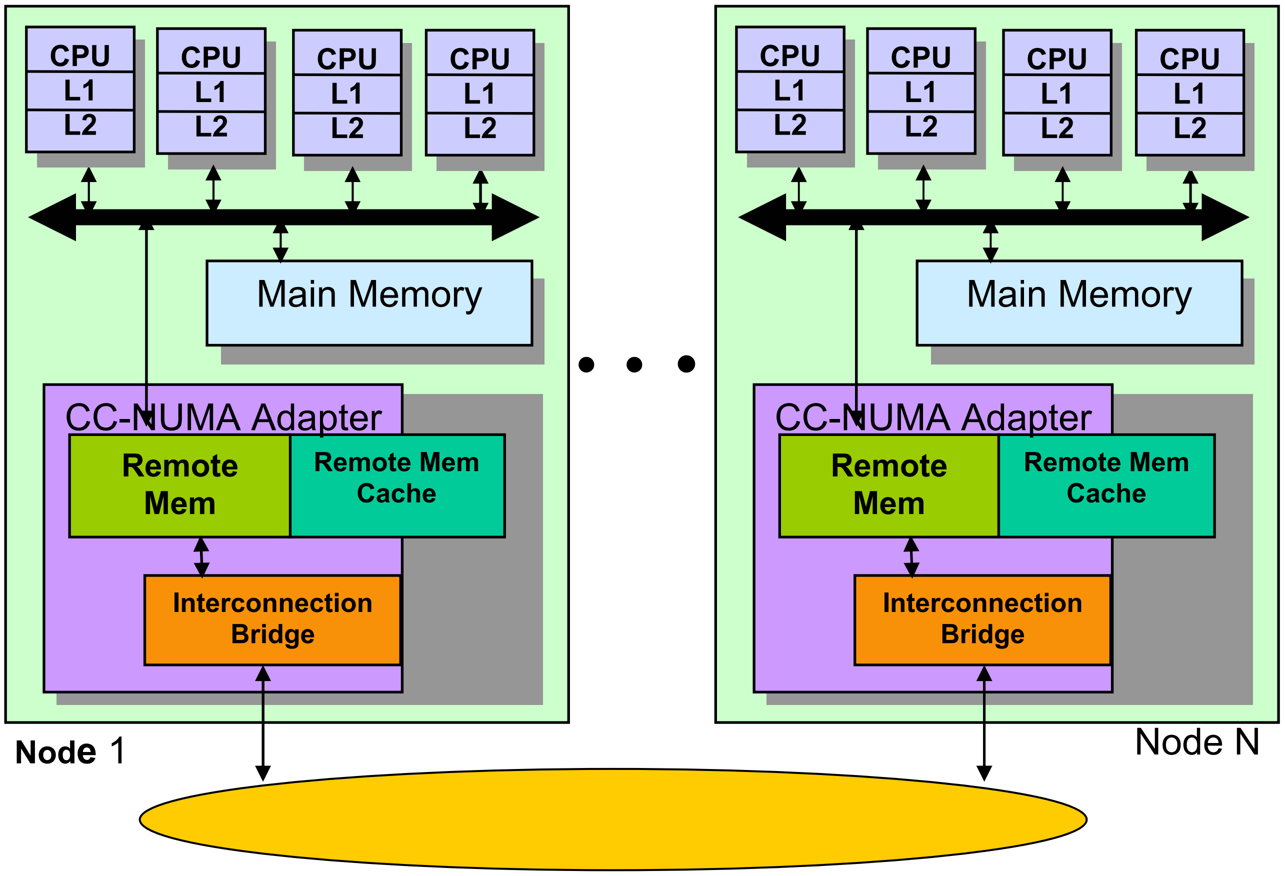
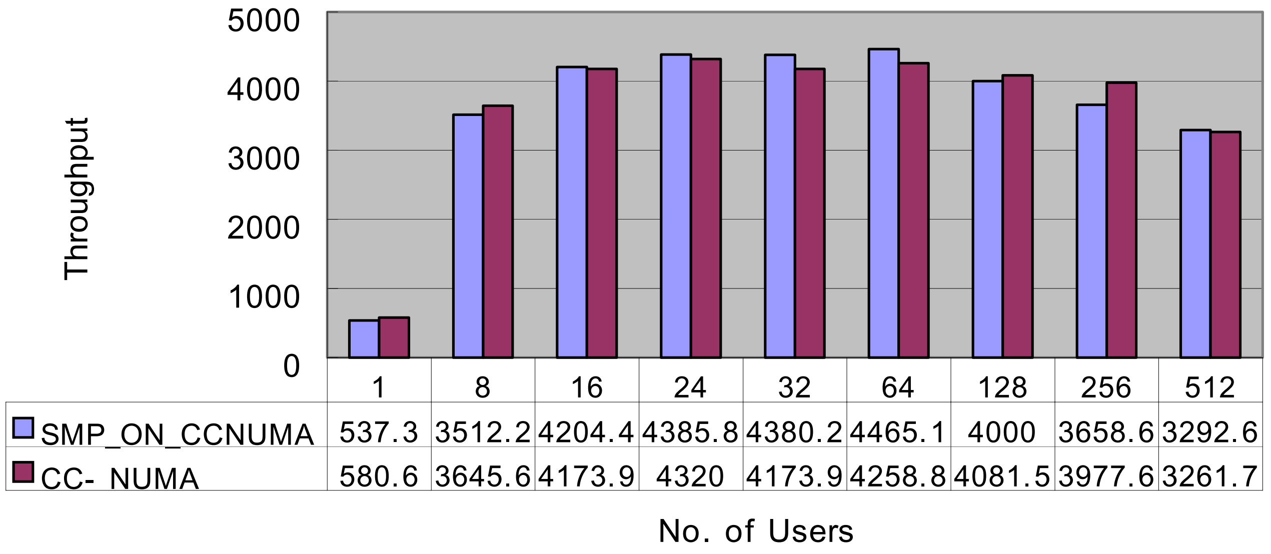
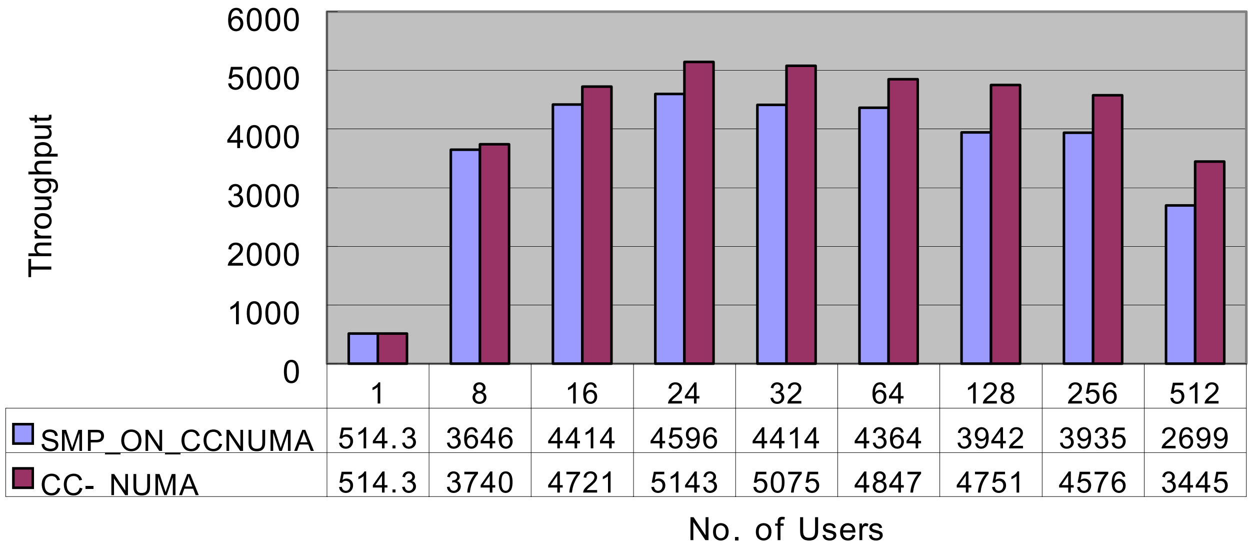


Fig 1. CC-NUMA Architecture



	1	8	16	24	32	64	128	256	512
🟦 SMP_ON_CCNUMA	537.3	3512.2	4204.4	4385.8	4380.2	4465.1	4000	3658.6	3292.6
🟥 CC- NUMA	580.6	3645.6	4173.9	4320	4173.9	4258.8	4081.5	3977.6	3261.7

Figure 2. SDET throughput (1 node)



	1	8	16	24	32	64	128	256	512
🟦 SMP_ON_CCNUMA	514.3	3646	4414	4596	4414	4364	3942	3935	2699
🟥 CC- NUMA	514.3	3740	4721	5143	5075	4847	4751	4576	3445

Figure 3. SDET throughput (2 nodes)

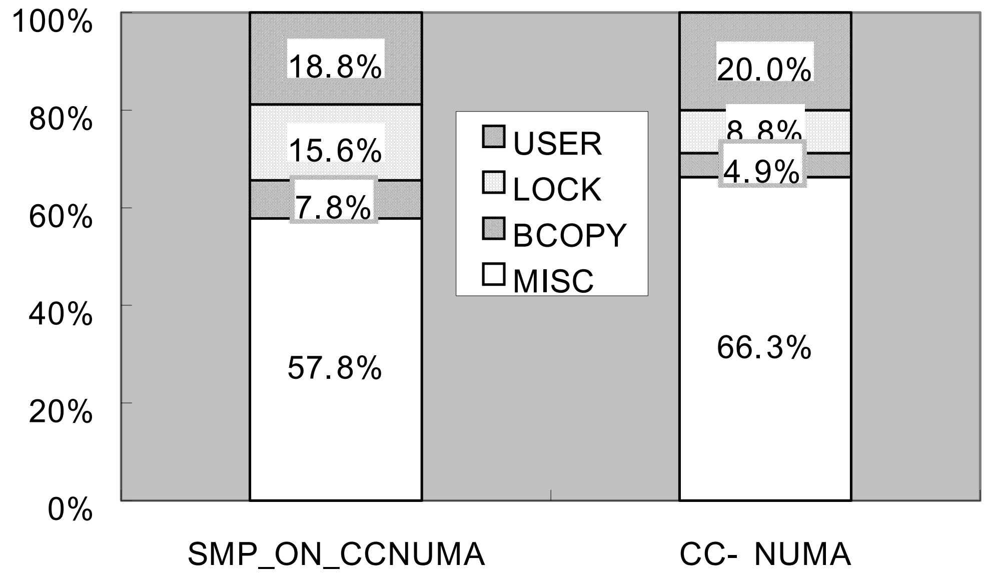


Figure 4. Kernel profile data for 2-node configuration

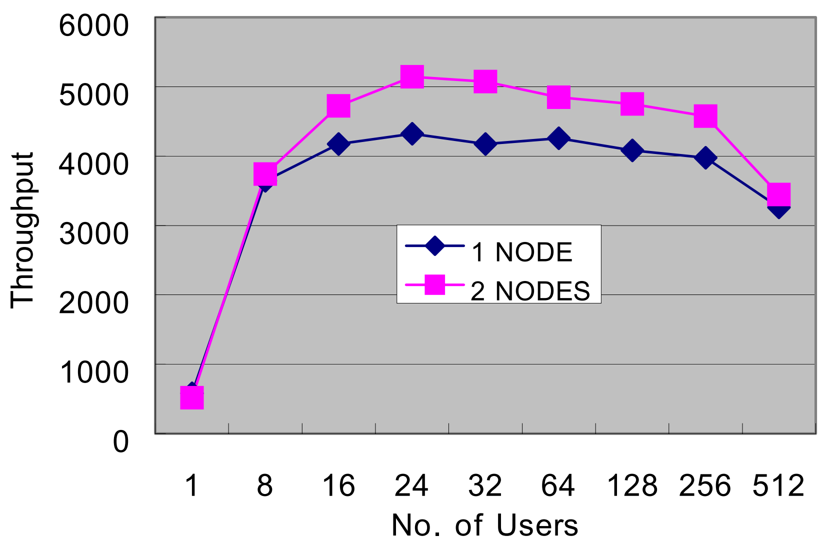
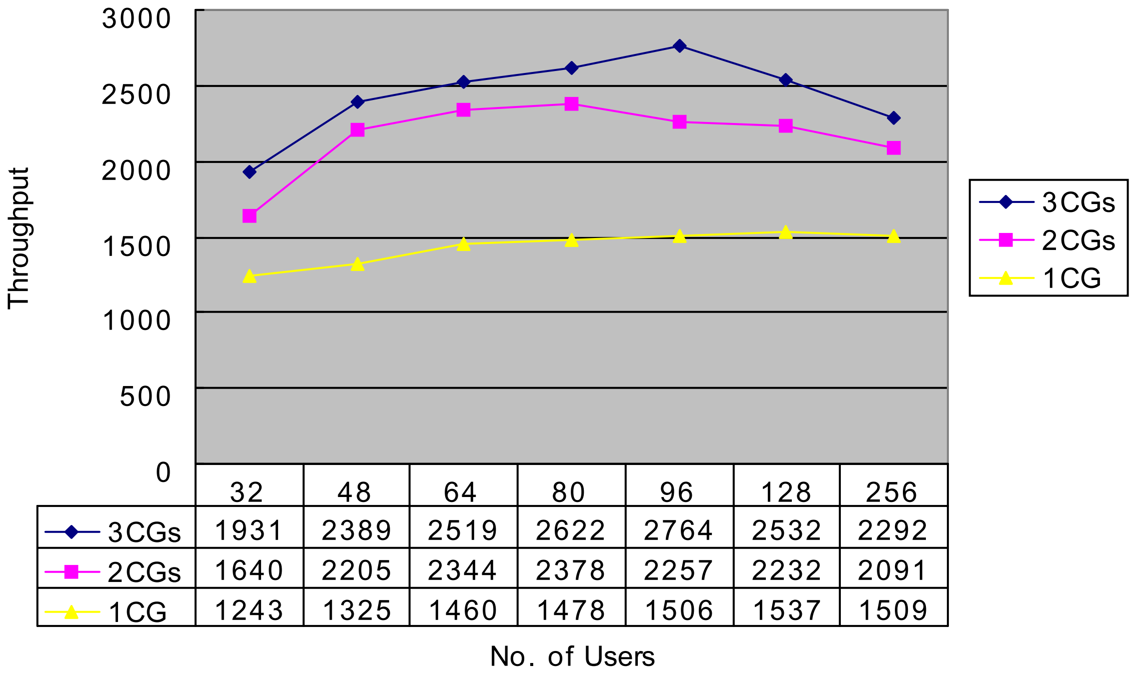
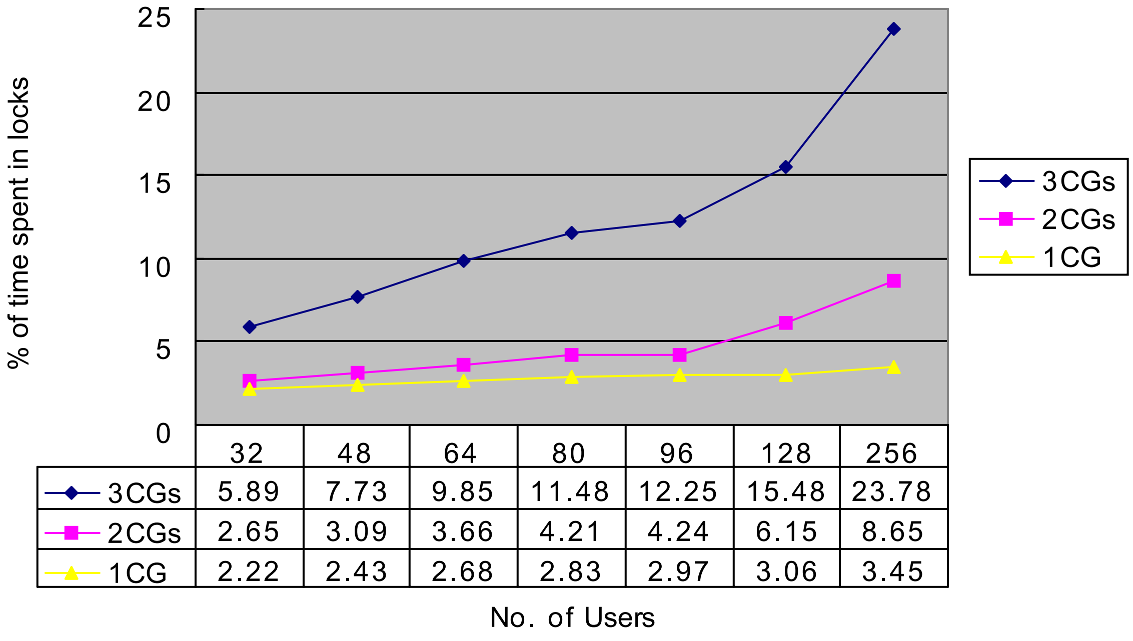


Figure 5. Scalability of CC-NUMA version



	32	48	64	80	96	128	256
◆ 3CGs	1931	2389	2519	2622	2764	2532	2292
■ 2CGs	1640	2205	2344	2378	2257	2232	2091
▲ 1CG	1243	1325	1460	1478	1506	1537	1509

Figure 6. Node scalability



	32	48	64	80	96	128	256
◆ 3CGs	5.89	7.73	9.85	11.48	12.25	15.48	23.78
■ 2CGs	2.65	3.09	3.66	4.21	4.24	6.15	8.65
▲ 1CG	2.22	2.43	2.68	2.83	2.97	3.06	3.45

Figure 7. Spin lock contention