SNA Server/6000 Performance on SMP

By Dov Bulka, Michelle Hermanson, Chris Selvaggi, and Julia Sime

This article describes how IBM achieved extremely high scalability during the port of the uniprocessor SNA Server/6000 to a Symmetric Multiprocessor (SMP) environment. Lessons learned should help other developers porting to an SMP environment.

AIX SNA Server/6000 is a high-function, high-performance communications stack for AIX. It supports legacy System Network Architecture (SNA) applications, such as 3270 emulators, LU0 home-grown applications, and LEN-level LU6.2, to enable connections to OS/400®, OS/2®, MVS™, and many other platforms. SNA Server/6000 supports Advanced Peer-to-Peer Networking® (APPN®) to reduce configuration complexities and overhead while maintaining reliability and robustness.

AIX 3.2.5 has supported SNA Server/6000 for several years. SNA Server/6000 2.2 became available several months ago in an AIX 4.1 Uniprocessor (UP) configuration. SNA Server/6000 3.1, which has efficient SMP performance, is now available under a beta-test program and will be generally available later in 1995.

Design Background

Before going through the porting process, let's look at the design before the port. Some design decisions made during the product's infancy worked well into our SMP plans; others did not. Some design features became major hurdles in our attempt to use the SMP configuration efficiently.

SNA Server/6000 consists of a set of user space daemons, kprocs, pseudo-device drivers, and callback functions provided to lower-level pseudo-device drivers. The number of processes active in SNA code can range from a dozen to thousands at any given time. The user space daemons provide all types of control and start/stop services. The kprocs, device drivers, and callback functions participate with the daemons in start/stop services. They also provide steady-state services on their own, completely within the kernel.

Once links and sessions have been established, only three threads participate in the steady-state flow of data over the network, as shown in Figure 1.

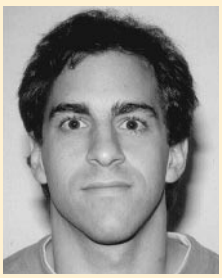
- **The Transaction Program (TP)** is an application that uses SNA Server. Provided by the customer or another product, the TP's thread calls the SNA Server interface that invokes the SNA Server/6000 device driver.
- **The Half Session (HS) kproc** provides data flow control and transmission control services at one end of a session. SNA Server provides one HS kproc for every active session.
- **The Data Link Control's (DLC's) kproc** is provided by the DLC pseudo-device driver.

These three threads—TP, HS, and DLC—are involved in sending data over the network. The TP thread, running in SNA Server's device driver code, puts the data into system memory buffers (`mbufs`) and queues it to HS. After performing its transport layer function, HS calls the DLC device driver interface. This, in turn, calls the adapter device driver interface, passing the data to the adapter while still on the HS thread.

When data is received from the link, the communications adapter passes the data to the bottom half of the DLC device driver, which quickly queues it to the DLC kproc. The kproc enters the bottom half of the SNA device driver and queues the data to HS. Once HS completes its processing, it then queues the data to the TP thread, which receives the data when it enters the read call of the SNA device driver. At this point, the


Dov Bulka


Michelle Hermanson


Chris Selvaggi


Julia Sime

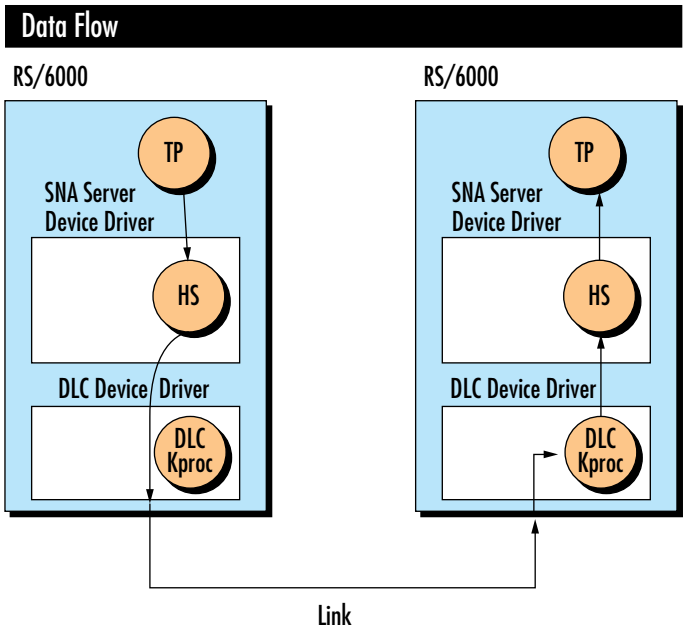


Figure 1. Steady state data flow

data transfer is complete, and the TP can process the data as it chooses.

Many of these data transfers can be active simultaneously. Memory shared between these active threads is protected by locks. This method of concurrency control functions correctly on both UP and SMP systems.

Although the code ran correctly on the SMP machine, we encountered a substantial performance problem. The early performance measurements showed that TP response time over 4-way SMP was worse than UP response time by a factor of 10.

Fortunately, the design did not require major changes to improve scalability; however, the changes made were not always easy or obvious. The changes included converting lock calls, refining lock granularity, shortening path length within locked sections, restructuring to eliminate excessive locking, reducing cache misses, and pooling memory. The primary benefit of these changes was to reduce contention for shared resources.

The Thundering Herd

Performance analysis showed that lock contention was a major cause of the initial poor SMP performance. Several critical locks had high *miss rates*—the result of a thread failing to acquire a lock, which requires the thread to wait. The high miss rates resulted from the `lockl()`—a blocking lock—system call used by SNA Server/6000. A blocking lock suspends the thread until the lock is free. When the lock is freed, all suspended threads are placed on the run queue to contend for the lock. When lock contention is heavy, this behavior is very inefficient.

In a distributed application, it is common to have several threads waiting for a critical lock. When a lock is freed, the suspended threads are signaled; all but one are put back to sleep, resulting in many wasted context switches. A specific thread bounces between the suspended and run queues before actually acquiring the lock. This problem is sometimes called the *thundering herd* because all the threads vie for the lock each time it is freed.[1]

AIX 4.1 provides a new type of lock called a *simple lock*, which can be used to solve the thundering herd problem. An important feature of the `simple_lock()` call is spinning. A spin lock allows the waiting thread to keep its processor, repeatedly checking the lock in a tight loop (spin) until the lock becomes free. This is ideal when locks are held for a short time. Spin locks avoid costly context switches. By converting to simple locks, we significantly reduced our lock miss rates. The move from `lockl()` to `simple_lock()` was the most important modification we made—giving us the largest performance boost on SMP.

The simple lock performance improvements introduced several new problems into the existing code: two benefits of `lockl()`—signal delivery and recursion—were not compatible with simple locks. Our code was heavily dependent on signal delivery to alert the kproc to conditions that require termination. The `lockl()` allows a thread that is waiting for a lock to receive signals; simple locks do not. To avoid deadlock on kproc termination, the code had to release all locks before it could signal a kproc.

Lock recursion or nesting is the other `lockl()` feature not available with simple locks. In SNA Server/6000, code modules are called from several different paths. When using `lockl`, these shared modules could lock a resource without worrying if the resource might already be locked on this path. If the lock was already held, it would simply be nested. To convert to simple locks, we analyzed each location to decide if the lock was already held. We restructured the code so that many nested calls were removed.

[1] Campbell, Mark et al. *The Parallelization of UNIX System V Release 4.0.* USENIX. (Winter 1991).

Other Lock Issues

After overcoming the thundering herd obstacle, we had more work to do to achieve high SMP scaling. Next, we refined locking granularity, reduced the time for which locks were held, and eliminated some locks.

First we eliminated the coarse grain locks from several places in HS. Instead of using one lock to protect a region of code, we protected the integrity of various shared data structures using locks specific to those data structures. Because unrelated structures maintained separate locks, they could provide simultaneous access. In a sense, we traded region locks for data locks.

We also reduced the time that locks were held. We converted linked lists to hash tables to speed up search, access, and update operations. Since those operations were critical code sections, they were protected by locks. The faster an operation completes, the faster the lock is released to allow the next operation to proceed.

Converting linked lists to hash tables significantly increased the speed of execution on UP and SMP configurations. It also reduced the instruction count—always a welcome change.

Time-consuming operations, such as copying application (TP) data from user space to kernel space, were removed from the scope of critical sections. This significantly reduced the length of time in the locked section of code. The combination of the short time during which locks were held and the use of simple locks enabled waiting threads to obtain the lock after a short spin and to avoid a costly context switch.

We eliminated some lock calls. For example, SNA associates each system `mbuf` (communication buffer) with its own structure—the `snaipcm`. A one-to-one mapping exists between a `snaipcm` and an `mbuf`. As the `mbuf` pointer travels along the data transfer path, it is passed as an argument from one routine to another.

A macro, `MBUF_SNAIPCM`, is used in these routines to retrieve the `snaipcm`, given the `mbuf`. This macro call was not a performance problem on AIX 3.2, because the `snaipcm` was stored in the `mbuf` itself, so the `MBUF_SNAIPCM (mbuf)` call translated to a simple pointer dereference.

Changes made in AIX 4.1 made it difficult, if not impossible, to embed the `snaipcm` in the `mbuf`. Since only the `mbuf` was passed from routine to routine, it would have been necessary to change hundreds of code paths to pass both the `mbuf` and the `snaipcm`. For the first pass, we redefined the macro to map the `snaipcm` to the `mbuf` using a hash table. Since several different processes could access this hash table concurrently, it required a lock. We measured the performance hit, and it was considered acceptable on a UP system.

The performance hit of the new lock calls was much greater on an SMP than on a UP system. With four processes running at the same time, lock contention for this hash table lock was very high. We eliminated this bottleneck by creating the `mbuf` to `snaipcm` mapping when the `mbuf` arrives, then passing the mapping around instead of the `mbuf` pointer alone.

We constructed a new structure for `mbuf` arrival that contained both pointers to `mbuf` and the associated `snaipcm`. This specific mapping occurred in hundreds of places throughout the code. Each modification demanded caution to prevent errors from being introduced into working code. In fact, we limited the scope of the modifications to code paths considered critical to performance. The result was a very significant reduction in the number of lock calls.

The number of processes active in SNA code can range from a dozen to thousands at any given time.

False Sharing

When unrelated entities share a memory unit such as a cache line, main memory frame, or a disk page, it is called *false sharing*. The sharing is false because the entities are functionally unrelated and their proximity is usually unintended.

Two "hot" lock variables were declared beside one another, and the compiler placed them accordingly. Consequently, both locks shared the same cache line. It was not uncommon for these two locks to be updated by different threads simultaneously. On an SMP machine, that creates a "cache consistency storm" in which each CPU invalidates the other's cache line. This results in wasted bus traffic while the hardware attempts to keep the caches consistent. To resolve this issue, we separated the lock definitions so each occupied a separate cache line—a typical cache line size on our RISC System/6000 platform is 32 bytes.[2]

Posting While Holding a Lock

SNA Server/6000 has several cases in which one process performs a task on behalf of another, which requires Interprocess Communications (IPC). Consider the example of the Half Session kproc, which transfers data over a link for a TP.

[2] Debora Blakely-Fogel. "Porting Applications to the AIX 4.1 OS SMP Environment," *AIXpert* (November 1994).

Tips for Porting to an SMP Environment

We learned the following lessons in porting SNA Server/6000 to SMP:

- Prevent the thundering herd by replacing blocking locks such as `lockl()` with spin locks such as `simple_lock()`.
- Eliminate locking if possible.
- Reduce the size of a critical section by moving unrelated code outside the scope of the critical section.
- Reduce the duration in which locks are held by implementing more efficient algorithms.
- Avoid false sharing.
- Do not post another thread while holding a lock; post after releasing the lock.
- Manage your own memory pools to reduce allocation time on critical paths.

To do this, the data is first placed on a queue under the TP thread. Next, HS is notified via the kernel service `et_post`. This system call is considered to be the fastest IPC method. It is so fast that the post recipient can be scheduled to execute almost immediately, especially on an SMP machine with an available processor. However, if the recipient must acquire a lock to access the data and the sender holds the lock, then `et_post` loses its efficiency. This was the case with HS and TP data. We resolved this by performing the `et_post` after relinquishing the lock, which enabled the HS to run unimpeded.

Memory Pooling

A *heap* is a pool of memory chunks of various sizes managed by the OS. When it is accessed via the kernel service `xmalloc()`, a system lock is held. SNA manages several of its own pools of same-size memory chunks, which has two important benefits:

- Avoids the overhead of contention on a system lock
- Avoids the path length required to manage the complexity of the system heap

We can return memory from a pool of same-size chunks with very few instructions. We chose memory allocated frequently on critical paths and converted from `xmalloc()` calls to our own internal memory pooling.

Results

Before beginning the development work, we benchmarked SNA Server/6000 performance on a simple network. The objective was to assess the scalability of our code on an SMP system and to estimate performance gains going from UP to SMP. To keep all other variables intact and focus on scaling issues, we used the same two RS/6000 Model J30s, both 4-way SMP machines. We measured first with only one processor enabled on each machine. Then we took the same measurements with all four processors enabled. Since the AIX 4.1 SMP kernel is estimated to be 15% slower than the AIX 4.1 UP kernel, the one processor configuration is only an approximation of UP performance.

To create high-volume traffic, we invoked 150 sessions in which a 2 KB data buffer was repeatedly bounced back and forth between source and target TPs. With that level of traffic in the background, we measured the response time of a 1 MB data transfer from the local to remote machine over a dedicated 16 Mbit/second Token-Ring connection. Since we had no interest in the performance of the DLC and the adapter, we separated the measured data transfer TP from the other 150 sessions by routing them over distinct DLCs. The data transfer TP ran over a Token Ring while the rest of the sessions ran over a different link. If all traffic had been run over a single communications link, the link would probably have become the bottleneck and negated the advantage of having multiple processors.

When we began our work, our performance was 10 times slower on four processors than on one processor. When we were finished, response time on the 4-way configuration was 3.3 times faster than the 1-way configuration. This is a scaling factor of 3.3 out of the theoretical maximum of 4.0 on a 4-way SMP. This significant speedup—from .1 to 3.3—results from multiple SNA Server/6000 threads executing simultaneously on multiple processors.

Future Plans

We did not implement all performance enhancements because of tight deadlines; some, such as frequently accessed hash table locks, were deferred. During development, we considered replacing the table locks with fine grain bucket locks that would lock individual buckets as opposed to locking the whole table. That would enable two distinct buckets to be accessed simultaneously. Even though this feature was deferred,

it is an important tool for SMP developers. It could become useful when contention for a table lock is high.

What about processor affinity? It would be good if threads would dispatch on the same CPU that they last visited. If a thread can keep running on the same CPU, chances are good that the cache still contains relevant data. Otherwise, the cache must be drained to make room for the current thread's data. Although the AIX 4.1 kernel attempts to help with processor affinity, it is not perfect.

We considered binding the HS kprocs to the various available processors. The disadvantage of this design would be that a kproc may occasionally remain on the run queue since its target processor is busy, even though other processors might be available.

Acknowledgments

Many people contributed to the SNA Server/6000 product family described in this article. Some design decisions made years ago played an important role in our successful SMP implementation. The authors would like to acknowledge the work of past and present members of the AIX SNA organization who are too numerous to list. Special thanks also goes to Herman Dierks from the IBM Austin Performance group, who contributed many valuable insights into SMP efficiency.

Dov Bulka, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: dov@ralvm5.vnet.ibm.com. Dr. Bulka is an advisory programmer working on SNA products for AIX. He has a BA in Mathematics from Brandeis University and a PhD in Computer Science from Duke University.

Michelle Hermanson, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: mmh@vnet.ibm.com. Ms. Hermanson is a developer for the SNA Server/6000 group and works primarily on kernel extensions and device drivers. She has a BS in Computer Science from the University of Michigan.

Chris Selvaggi, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: zamboni@vnet.ibm.com. Mr. Selvaggi, a senior associate programmer, is a member of the AIX SNA team developing SNA Server/6000 and AnyNet/6000 Sockets over SNA. He has a BS in Computer Science from the Georgia Institute of Technology.

Julia Sime, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: julia@vnet.ibm.com. Ms. Sime is a development programmer working primarily on kernel extensions and device drivers using C and C++. She is currently the chief programmer for the AIX SNA Server/6000 Version 3.1. She studied computer science at California Polytechnic University in San Luis Obispo, California.

POWER SPOT

High-Performance World Wide Web Server for 1996 Olympic Games in Atlanta

Want to know what's happening at the Atlanta Olympics next summer? Now you can surf your way to a world of information on the 1996 Olympic Games World Wide Web Server at the URL below:

http://www.atlanta.olympic.org

This new server will provide continuously updated facts, photos, videos, and audio content—all aimed at providing the latest possible news on Olympic events.

Many Internet users from around the world are expected to access the new Olympic information site. In order to handle this enormous workload, the site will use IBM's high-performance RS/6000 Scalable POWERparallel (SP) system as a server. As the volume of online Olympic Games information expands, the SP server can scale up easily—with additional processors, memory, and disk storage—to accommodate the additional processing and data needs.

The RS/6000 SP will interface with IBM ES/9000™ and AS/400 systems, which run the sporting event results and information/communications applications for the 1996 Games. Working together, the systems will offer Internet users access to real-time news and information about the 1996 Games.