Turning the AIX Operating System into an MP-capable OS

Jacques Talbot
Bull

Abstract

This paper describes those MP features that Bull and IBM together introduced into the AIX operating system to support the Symmetric Multiprocessor machine marketed by Bull under the Escala name and by IBM under the RS/6000 Models G30, J30 and R30 names. The PowerPC architecture and the AIX operating system present some specific challenges. We present the major problems encountered and how they were solved.

1. Introduction

This paper is quite broad in the sense that changing a sophisticated UNIX implementation such as the AIX operating system into an MP-capable system is a complex task. So we have tried to briefly describe the problems encountered and then to provide a more detailed description of some of them, which were specific to the AIX operating system and the PowerPC.

2. The Hardware Architecture: a summary

Only the HW features which have a significant impact on the SW will be described here. The architecture is a so-called SMP, Symmetric Multi Processor based on PowerPC [IBM94]. An SMP is a system where several CPUs share a common memory address space and I/O subsystem. There is no memory local to CPUs (except caches which are almost invisible to SW). The Symmetric attribute is perhaps not the most significant one since most machines in the MP category to-day are symmetric, in the sense that all processors get equal access to all I/Os. The exceptions to this rule are primarily low-end Intel-based MP servers derived from PC technology or machines dedicated to specific needs (e.g. real-time).

So SMP is better described as "Shared memory with HW coherency". It means that all CPUs see the "good old programming model" of a single memory with a more or less uniform access time, i.e. not depending on the addresses accessed. Moreover, all CPUs, when reading the "same" address, get the same value. In presence of copy-back caches, it is not so easy to achieve this simple property; this is the purpose of the HW cache coherence protocol, named according to the MESI acronym which designates the 4 possible states of a cacheline in the state machine (Modified, Exclusive, Shared, Invalid). The SW can safely ignore this complexity, assuming that the HW is properly functioning.

The same characteristics of uniform access time and coherent memory allow to run a single copy of the operating system on the machine. So all CPUs can share a single queue of ready-to-run execution threads, resulting into better system load balancing. Getting applications for the platform is pretty simple. There is actually no porting activity in the general case, since applications can rely on the traditional assumptions that they have made since computers are with us. Binary compatibility with the uniprocessor applications is the rule.

In contrast, architectures such as MPP (Massively Parallel Processors) and other kinds of tightly coupled clusters, where the above two characteristics are not present, generally require that applications have some knowledge of the HW architecture.

2.1 PowerScale architecture

This specific SMP architecture is called PowerScale. There are up to 8 PowerPC (601, 604 or 620) CPUs accessing a common memory through a cross-bar for data and a bus for addresses. A cross-bar is a data exchange device modeled after a telephone cross-bar switch where each agent has a private data path to the interconnection HW and multiple connections can occur simultaneously. An I/O bridge allows access to two MCA busses.

The PowerScale architecture is optimal from a price-performance standpoint for the target we were shooting at, i.e. a mid-range SMP optimized for 4 CPUs, still efficient at 8 CPUs and sellable in monoprocessor.

2.2 Caches

Each of the 8 PowerPC CPUs has a (relatively) small internal (32 or 64 Kbyte) Level-1 cache and a large external (1 or 2 MByte) Level-2 cache. Each cache is organized in "lines". Data transfers and coherency updates take place in unit of lines. A cache line is 32 or 64 bytes wide, depending on the PowerPC model. Both caches are copy-back (sometimes called write-back), meaning that when a write is performed, the modified value is written only in the cache, and pushed in lower memory hierarchy levels only when required, typically when the "dirty" cache line is needed for other data.

The other option is write-through, where main memory is updated on each write. This technique is much simpler but considerably less efficient.

In a copy-back cache, when several copies of a "clean" (unmodified) line exist in several caches, and one of these is modified, there are two options for other copies: update or invalidate. Write-invalidate is best from a performance standpoint for data which will probably not be used in the near future by the CPUs where the line is invalidated. Write-update is best when the probability of reuse is high. In most cases, for UNIX applications and kernel pages, write-invalidate is the preferred mode since most lateral misses (cache to cache) are related to a working set migration which is a one-way event and seldom associated with real data sharing. Since the PowerPC architecture implements only write-invalidate, there is anyway no choice to make.

It is possible, HW-wise, to tag pages as cached (copy-back or write-through) or uncached. Except for some phases of bootstrap in AIX, all memory is tagged as cached and copy-back.

2.3 Atomic operations

To construct the locks that SMP SW needs, an atomic operation is needed. It is not the traditional *test_and_set* or *compare_and_swap* instructions of CISC processors that are used. The PowerPC, like many RISC architectures, implements *load_and_reserve* and *store_conditional* instructions such that if a reservation bit is "broken" between the Load and the Store operations, the Store fails. These two instructions allow the efficient implementation of all the atomic operations that concurrent SW needs, including *test_and_set* and *compare_and_swap*.

2.4 Memory order model

Until recently, UNIX multiprocessors had a *strongly ordered* memory model. This means that the order of loads and stores as seen by the program cannot be changed between the output of a CPU and the memory system, where the other CPUs see the effect of these operations. Modern architectures are *weakly ordered*. The HW is allowed to reorder loads and stores between the CPU and the memory. This is done to optimize performance, when the first operation has to go to main memory for completion while the second one can be rapidly *globally performed* in the Level-1 cache. In that case, the second is globally performed before the first. Of course, the order of operations as seen by one CPU is not changed, because the HW keeps track of data dependencies and always provides the latest value of a data to the CPU, even if not yet globally performed. Only the visibility of these operations from the other CPUs is impacted. This has some importance only when CPUs synchronize or communicate with one another, because one CPU could get stale values for some data.

A similar issue results from aggressive prefetching of data (*speculative execution* found on 604 and 620) which allows a CPU to read some data very much in advance, and even across a test-and-branch sequence. This again allows a CPU to read data which has stale value.

Special instructions (*isync* and *sync*), also called import and export fences, allow to solve these problems. However, inserting these instructions stalls the pipelines and is therefore a performance drain.

2.5 Interrupt handling

The HW allows interrupts from one bus adapter to be addressed either to a specific CPU or to a set of CPUs. This set actually designates "one among all CPUs" and this CPU is dynamically chosen by the HW at each interrupt from a set of registers, one register associated with each CPU. Each CPU writes in this register a value indicating its willingness to receive interrupts.

3. Some high-level SW issues

Let's now jump abruptly to a very different level of abstraction. The main issue to transform a UniProcessor (UP) Operating System like the AIX operating system Version 3 into an MP Operating System like AIX Version 4 is to protect the coherency of the data structures managed by the kernel.

This is because several CPUs can simultaneously manipulate these data structures. The conflict can be a thread-thread conflict or a thread-interrupt conflict. On a UP machine, the thread-thread conflict cannot exist since only one flow of execution exists at any time (true only for a non-preemptable kernel); the thread-interrupt conflict is solved by masking the processor against interrupts. In an MP OS, locks must be added.

Apart from becoming an MP OS, the AIX operating system was also turned from a process-based kernel to a thread-based kernel. Several threads of execution exist within the scope of a process which is now a passive entity representing an address space and access rights. Threads are especially useful for MP architectures since they enable finer-grain parallel programming, compared to current multi-process applications; this gives an opportunity to have a better speed-up on multi-threaded applications, as opposed to the scale-up (more throughput) which can be obtained without parallel programming. A single multithreaded application will run faster for a single user on an SMP machine.

Threads-based programming requires a new API (POSIX 1003.1c) and thread-safe libraries. It opens the way to portable thread-based applications. We will not describe threads in more detail in this paper.

Turning a UP UNIX kernel into an MP kernel has been described several times during the last years in various USENIX proceedings [Cam91] [Car93] [Pow91]. We will therefore try to focus below on AIX-specific or PowerPC-specific issues.

4. Locks

4.1 Some interesting AIX Version 3 features

AIX Version 3 has two specific properties which affect the MP process.

- preemptability: when an interrupt occurs that makes schedulable a thread which has a higher priority than the currently running thread(s), a context switch is required. If the current thread is running in user mode, there is no issue. It is temporarily suspended, and this process is called preemption. However, if the current thread is running in kernel mode, most UNIXes do not allow the preemption to occur. This is to prevent corruption of the kernel data structures. The context switch is delayed until the current thread goes back to user mode. Since this can last quite a while on long kernel operations, this behavior is detrimental to response time and real time requirements.

  There are two possibilities to overcome this: preemption points and full preemptability. Preemption points are the poor man's solution, meaning that the kernel is not preemptable except at some specific points. Full preemptability means that the kernel is always preemptable except during a few code sequences. AIX Version 3 is fully preemptable. This implies a locking scheme to maintain data coherency.

- pageability: the AIX kernel code and data structures are pageable, except for some parts pinned in memory to avoid deadlock scenarios. This minimizes the amount of real physical memory required by the kernel by paging-out seldom used code and data. It allows to over-dimension kernel tables so that the hideous error message "too many processes or messages or whatever" encountered on traditional UNIXes is never seen on AIX. When a demand peak for a resource occurs, one or more pages of the corresponding table are paged-in and used. When no longer required, they are paged-out.

  So page faults and their associated context switches can be encountered at any point in time when kernel code runs. This is another case of unexpected kernel preemption.

To implement the two properties above, AIX Version 3 has a simple locking scheme with a coarse granularity; this scheme cannot be used for an MP system because the coarse lock granularity results in a very high contention rate on the few locks and a very poor overall scalability versus the number of processors. Processors spend their time waiting for the locked resources. However, the V3 locking scheme provided pointers to the critical areas that needed to be addressed.

These two AIX features obviously had to be kept for the MP version. It should be noted that the refinement of the locking scheme adopted for the sake of MP efficiency had the side effect of improving the preemptability of the kernel. In fact, tuning the locks for low contention tends to reduce the pathlength of the code spent with preemption off.

4.2 What should be locked?

Two basic approaches to locking exist:

- code-based locks are coarse grain locks which encapsulate large pieces of code and associated data. Basically, when entering a subsystem, a lock is taken and released only at subsystem exit. This is simple but collision rates can be high since such locks are taken for a long time.
- data-based locks are associated with data structures, or part of them, and lock-unlock operations are inserted in the code so that data protected by the lock are kept coherent when the lock is free. Also data-based locks can be coarse-grained (a whole table), medium grained (some elements of a table) or fine grained (one element of a table, or even a single item in a structure).

Both types of locks are used in AIX Version 4. The choice of lock type is dependent upon how critical to performance is a given subsystem. For example, the CD-ROM filesystem has few requirements for simultaneous access due to the physical constraints of the device. So a code-lock can be used.

On the other hand, the table of messages for System V IPC must be fine-grained because this is critical to the aggregate message switching capabilities of the system, in presence of several message queues, readers and writers.

4.3 Which type of locks?

Many lock semantics can be imagined. Because we wanted to be able to borrow some code from the OSF/1 kernel, and because of the experience Bull had accumulated with the MP aspects of OSF/1, we decided to model our locks on the OSF/1 kernel locking scheme.

In the case of a resource being unavailable, the requester thread blocks. This can be either by spinning (sometimes called "busy wait") or by releasing the CPU and going to sleep. Sleeping is not allowed if called from an interrupt level because it leads to deadlock situations.

Two types of locks are used:

- simple locks: as the name implies, this is the most rudimentary form and as a result the most efficient and frequently used. The lock is a simple word and if busy, the requester spins. However, the lock implementation has been optimized as follows:
  - — After some amount of spinning, the requester of a busy lock will sleep (except if the request is made from a interrupt-level or masked against interrupts). The appropriate spinning time is approximately the time spent to execute a process switch.
  - — If a lock is busy and the owner of the lock is not running on any of the CPUs, the requester goes to sleep. This is because the probability that the lock is freed in the spinning time-frame is almost zero.

    The owner of a lock is not running either because it is runnable but not currently running on a processor, or because it is waiting for some event (e.g. page fault resolution).
- complex locks: these are multiple-readers, single-writer semaphores. They can optionally be tagged as recursive. Recursivity is generally discouraged in the locking scheme to keep it simple and deadlock free. However, to be able to import recursive code, this option is left open to the programmer. Threads requesting a busy complex lock spin and then sleep just like requesters of simple locks.

  They are used only if Reader-Writer semantics make sense (e.g. in the filesystem) or if recursivity is needed.

Both types of locks can create the priority-inversion syndrome. This happens when high priority threads wait for a resource held by a lower priority thread. It results in a degradation of the response time. To overcome this, the priority of the thread owning a lock is temporarily raised to the priority of the highest priority thread waiting for the lock. This is sometimes called "priority promotion".

4.4 How many locks, how often?

A system may have many or few locks. It may use them often or seldom. In a first approach, fine-grain parallelization implies many locks, taken often. Coarse-grain parallelization has few locks seldom taken. Obviously, there is a continuum in a two-dimensional space. It is not always obvious to characterize a locking scheme.

At first sight, it looks that fine-grain locking is always better. It results in very short time spent while holding a lock, so low probability for collision (finding a lock busy). However, locking does not come for free. As opposed to older architectures where writing a word and accessing a lock through a *test_and_set* instruction were in the same ballpark in terms of execution time, there are orders of magnitude of difference between reading or writing a word in Level-1 cache (1 cycle) and taking a lock (around 50 cycles in the best case). The high number of cycles associated to locking is due to several factors, in order of importance:

- need to synchronize the pipelines on lock operations
- cache misses: processors are much faster these days, but memory does not keep up.
- SW overhead: for various reasons and despite all the tuning, several bookkeeping instructions are needed around the atomic core

So too much locking and unlocking is detrimental to performance. Instead of taking a lock for the minimal amount of time needed from a data structure coherency standpoint, the programmer has to consider whether the lock will not be needed again soon, and keep the lock during this interval, even if not strictly needed. The same applies to whether it is better to have several locks for a set of data structures and play subtle games or have a single lock held for a longer time but toggled less often.

So locking the kernel is finding a trade-off between too few locks and too much contention or too many locks and too much overhead. In both cases, performance suffers. There is an optimum but finding it is a matter of calculations, experience and finally trial-and-error process [Cam91].

Some figures below give an idea of the locking scheme complexity in AIX version 4. They must be considered as orders of magnitude more than exact figures. There are around 200 classes (types) of locks. The actual number of instances of locks depends on the size of various tables and is not very significant. The number of lock operations (statically counted) is around 500. The number of locks operations performed per second on a timesharing workload (SPEC SDET) is around 10,000 lock-unlock pairs per second, on a quad machine with 75MHz 601 processors.

4.5 How to avoid locks

Some parts of the kernel fit into well defined frameworks and in that case the burden of locking can be delegated to the framework code.

- Drivers which have no strong performance requirements can be "funneled". In that case the drivers framework locks the driver on a "master" processor and all accesses are serialized via a single lock. This scheme is used only for third party-devices. None of the Bull or IBM-developed drivers use it.
- The streams framework [Rit84], taken from OSF/1, can provide locking at various levels, described below from the highest level of concurrency to the the lowest level :
  - — queue level: only access to message queues is protected. Modules must provide their own locking for shared state or upstream - downstream synchronization.
  - — queue pair level: both upstream and downstream queues are protected. If some data are not "per-stream", i.e. they are shared between several streams, the module must take care of their locking. This is used by TTY line disciplines, which have only per-stream data.
  - — module level: all queues and shared data associated with a module are protected. This is the default mode which can be safely used by most UP streams modules, which want to ignore MP issues.
  - — arbitrary level: allows to arbitrarily group together from the locking standpoint modules which share data other than via queues.
  - — global: one single lock for the whole streams subsystem, for debug purpose only

In AIX Version 4, the TTY subsystem, SNA and Netware stacks are implemented within the streams framework.

4.6 Deadlocks

As soon as we have more than one lock in a system, a potential for deadlock exists. One well-known strategy to avoid deadlocks is to implement a lock hierarchy, so that locks are always taken in the same order. We chose to implement a partial hierarchy, i.e. not global to the whole kernel but per subsystem. This hierarchy is defined in the registration process (see below under *Lock instrumentation*). However, it is not global nor enforced dynamically for reasons similar to the ones explained in [Pac91]:

- — strict ordering is hard to enforce globally and not always necessary, if the programmer knows that the deadlock cannot occur for some reason.
- — impact on lock code pathlength and therefore performance
- — desire to be able to import code from OSF/1 without major restructuring

We chose instead to use a static deadlock analyzer, called SDLA (Static DeadLock Analyzer). This tool processes kernel code to detect potential deadlocks by exploring the locking scheme tree [Kor89]. It is derived from *lint*. SDLA does not use the static description of the lock hierarchy but constructs its own vision of the lock hierarchy while walking the tree. Two common problems found in many similar tools are the "noise" (false deadlocks) and the time taken to explore the tree and perform the analysis. We found several techniques to overcome these in a practical manner (patents pending) and we were able to discover several kernel design errors. We use the SDLA to continuously inspect the source tree to ensure that the addition of features and correction of bugs do not introduce new deadlocks.

We foresee an enhancement to SDLA to be able to detect underlocking. Underlocking happens when data normally protected by a lock are manipulated without the lock being held. This can lead to kernel data corruption and system crashes. It is the most critical area for an MP kernel, since deadlocks are less difficult to debug. This will be similar to WARLOCK [Ste93].

4.7 Lock instrumentation

A comprehensive lock instrumentation is key to the kernel tuning process [Car93]. A symbolic naming and registration scheme has been adopted for all locks. This facilitates lock designation by the analysis tools. Locks are named as:

Subsystem_name$Lock_family_name$Occurrence_number

We implemented two levels of lock instrumentation:

- One is almost always on during validation phases (selection at bootstrap time) and records the number of acquisitions, misses and sleeps. The overhead has to be as low as possible. The current implementation adds 10% to the lock-unlock pair execution time. A tool named *lockstat* allows the customer or field-service to analyze the behavior of kernel locks.
  Below is a display of *lockstat* output, taken during the run of a database benchmark early during development.
  The 20 locks with largest `%Ref` are shown:

```
Subsys  Name                   Ocn   Ref/s    %Ref   %Block  %Sleep

PROC    PROC_INT_CLASS          -1    8434   29.89    18.77    0.00
IOS     UPHYSIO_LOCK_CLASS      -1    2483    8.80     6.74    0.00
PMAP    PMAP                     0    2108    7.47     1.60    0.00
DISK    SCDISK_LOCK_CLASS       -1    1495    5.30     4.22    0.00
IPC     SEM_LOCK_CLASS           2    1472    5.22     7.75    0.00
LOCKL   LOCKL                    7    1035    3.67     0.00    0.00
IOS     IOS_IPOLL_CLASS          2    1030    3.65     0.48    0.00
PFS     JFS_LOCK_CLASS          -1     997    3.53     1.46    0.00
XLVM    LVM_LOCK_CLASS           0     996    3.53     3.24    0.00
IPC     SEM_LOCK_CLASS           2     569    2.02     4.12    0.00
IPC     SEM_LOCK_CLASS           2     524    1.86     3.73    0.00
IPC     SEM_LOCK_CLASS           2     523    1.86     3.90    0.00
VMM     VMM_LOCK_SCB           443     499    1.77     1.66    0.00
IPC     SEM_LOCK_CLASS           2     475    1.68     3.49    0.00
PROC    TRB_LOCK_CLASS           3     445    1.58     0.05    0.00
PROC    TRB_LOCK_CLASS           2     443    1.57     0.03    0.00
PROC    TRB_LOCK_CLASS           0     441    1.56     0.02    0.00
PROC    TRB_LOCK_CLASS           1     438    1.55     0.04    0.00
TCPKER  DEMUXER_LOCK_FAMILY  45172     417    1.48     3.08    0.00
PROC    TOD_LOCK_CLASS          -1     399    1.42    12.23    0.00
```

`Subsys Name Ocn`	designate the lock.
`Ref/s`	is the number of requests for this lock per second.
`%Ref`	is the % of requests for this lock versus all lock requests
`%Block`	is the % of blocking (spin or sleep) requests versus all requests for this lock.
`%Sleep`	is the % of blocking requests resulting in sleeping versus all requests for this lock.

- For a more detailed lock analysis, we have a trace-based instrumentation, which can be turned on-off dynamically. We use the AIX trace tool and insert trace hooks in lock code. This allows us to compute the time spent under locks and get a comprehensive view of lock behavior. The LCA (Lock Contention Analyzer) is a Motif-based tool giving a complete picture of all lock parameters. It is used only in the development organization. Below is a display of *LCA* output, taken during the run of a database benchmark early during development.
  The 20 locks with largest `Coll Sec` are shown:

```
                          TABLE Collision Section (times in 10 s)
+--------------------+--------+---------+--------+------------+----------+---------+---------+
|              Class |    Ocn |    Type |  Count | Coll Sec(%) | CPU (ms)  | CPU Avg | CPU Max |
+--------------------+--------+---------+--------+------------+----------+---------+---------+
|     PROC_INT_CLASS |     -- | Simple  |  93579 |      6.108 | 2885.994 | 30.84   | 1712.00 |
|     SEM_LOCK_CLASS |      2 | Simple  |  15367 |      2.816 | 1330.436 | 86.58   | 2323.46 |
| UPHYSIO_LOCK_CLASS |     -- | Simple  |  25658 |      2.466 | 1165.270 | 45.42   | 1609.22 |
|  SCDISK_LOCK_CLASS |     -- | Simple  |  15397 |      2.039 |  963.444 | 62.57   | 1336.19 |
|     LVM_LOCK_CLASS |      0 | Simple  |  10265 |      1.552 |  733.498 | 71.46   |  576.38 |
| DEMUXER_LOCK_FAMILY|  45172 | Simple  |   4267 |      1.132 |  534.809 | 25.34   |  365.82 |
|    IOS_IPOLL_CLASS |     30 | Simple  |   2580 |      1.114 |  526.462 | 04.05   |  546.30 |
|              LOCKL |     45 | Lockl   |  13043 |      1.071 |  506.051 | 38.80   | 1790.21 |
|    IOS_IPOLL_CLASS |     27 | Simple  |   2383 |      1.049 |  495.487 |  07.93  |  545.28 |
| DEMUXER_LOCK_FAMILY|  41076 | Simple  |   3783 |      0.967 |  456.725 |  20.73  |  518.27 |
|     SEM_LOCK_CLASS |      2 | Simple  |   5899 |      0.805 |  380.185 | 64.45   | 1300.61 |
|     SEM_LOCK_CLASS |      2 | Simple  |   5231 |      0.719 |  339.934 | 64.98   | 1806.98 |
|     SEM_LOCK_CLASS |      2 | Simple  |   5407 |      0.713 |  336.979 | 62.32   | 1276.16 |
| DEMUXER_LOCK_FAMILY|  61556 | Simple  |   2104 |      0.673 |  318.065 | 51.17   |  309.89 |
|     SEM_LOCK_CLASS |      2 | Simple  |   4800 |      0.654 |  309.150 | 64.41   | 1171.71 |
| DEMUXER_LOCK_FAMILY|    116 | Simple  |   1937 |      0.608 |  287.322 | 48.33   |  318.59 |
|       VMM_LOCK_SCB |    489 | Simple  |   5234 |      0.559 |  264.332 | 50.50   |  387.33 |
|     JFS_LOCK_CLASS |     -- | Simple  |  11736 |      0.373 |  176.423 | 15.03   | 1192.45 |
|      U_TIMER_CLASS |     41 | Simple  |  13639 |      0.366 |  173.051 | 12.69   |  581.76 |
|    IOS_IPOLL_CLASS |      2 | Simple  |  11182 |      0.302 |  142.643 | 12.76   |   47.62 |
+--------------------+--------+---------+--------+------------+----------+---------+---------+
```

`Class Ocn`	designate the lock
`Type`	is the lock type: *simple* or *R/W* or *Lockl* (AIX V3 backward compatibility)
`Count`	is the number of lock requests during the period
`Coll Sec(%)`	is the collision section, i.e. the % of CPU time when the lock is held
`CPU (ms)`	is total CPU time (in ms) spent with the lock held during the period
`CPU Avg`	is average CPU time (in microseconds) spent with the lock held during one locking (i.e. between one lock and the corresponding unlock)
`CPU Max`	is maximum CPU time (in microseconds) spent with the lock held during one locking

The data above were taken during the development process and do not reflect the behavior of the shipped system. For example, at that point in time, `%Sleep` was always 0 because the locks were tuned to spin for a very long time. A lock in the process management (`PROC_INT_CLASS`) was a point of contention. This is clear from the `%Ref` and `%Block` in the *lockstat* display or the `Coll Sec(%)` in the *LCA* display. This lock is both taken often (`%Ref`) and often missed (`%Block`). With this information, this problem was taken care of in the MP tuning process.

Precedence rules among locks in a subsystem are registered in a database. This allows the LCA tool to detect rule violations and issue warnings.

4.8 Debugging

The identification of the thread owning a lock is systematically registered so that the deadlock detector contained in the crash analyzer can provide extended information. This is true even in production kernels.
During the development phase, locks can be compiled with additional sanity checks turned on so that "asserts" can be used. They are however too costly to go into production.

The symbolic kernel debugger has been enhanced so that when a CPU encounters a breakpoint, the other CPUs are also stopped through the CPU to CPU communication channel (MPC, see below). It is then possible to switch the debugger from one CPU to another for CPU-specific data.

4.9 Testing

For each subsystem, we have a set of tests specifically targeted to lock stressing. With the *lockstat* tool, we are able to ensure that the collision rate is high enough on each and every lock so that we are sure that they are appropriately stressed, looking either for overlocking ($\rightarrow$ deadlocks) or underlocking ($\rightarrow$ race situations leading to data corruption). The collision rate is also randomly varied between maximum value and 0 to explore windows of vulnerability.

4.10 Kernel Implementation issues

The price of locks After carefully coding lock and unlock primitives for simple locks, in assembly language, it was determined that the lock-unlock pair price in the lock free case was around 100 cycles (on a PowerPC 601). Notice that the number of instructions is much smaller. Some lock instructions (e.g. *sync*) are costly in terms of cycles because the 601, as any sophisticated processor, needs to flush its read and write queues.

MP/UP overhead Locking and unlocking brings an overhead compared to a UP kernel. We took the goal to limit this overhead to 5% on the throughput of macro benchmarks like the TPC family or SPEC SDM and LADDIS.
The core of the AIX kernel (i.e. not counting the drivers and loadable kernel extensions like NFS) actually exists in two versions derived from a single source tree. In the MP version, all locks are enabled. In the UP version, only locks mandatory to allow pageability and preemptability are enabled using *#ifdef*. So, the UP systems do not incur the MP performance penalty. The appropriate kernel is automatically selected at bootstrap time.

Thundering herd problem This occurs when several threads are queued waiting for a resource, the resource is freed and several waiting threads are awakened at the same time. For simple locks, this is avoided by selectively waking only the highest priority sleeping thread. For complex RW locks, the highest priority writer is awakened, or all the readers if no writer is waiting.

Interrupts and locks On an MP system, to protect thread-interrupt critical sections, it is necessary to take both a lock (for protection against other CPUs) and to mask interrupts appropriately (for protection against your own CPU). For performance and readability reasons, we packaged these operations together into two primitives: *disable_lock()* and *unlock_enable()*.
AIX Version 3 already had two levels for interrupt handling. Part of the interrupt code is executed at the HW interruption level. When HW critical management has been completed, the bulk of the interrupt processing, if necessary, is handled "off-level" at a lower SW interrupt level.
Since the AIX Version 4 kernel is thread-based, there was an opportunity to transform interrupt handling into normal thread-level code [Pow91]. We decided not to do so because of the performance overhead of thread level handling versus off-level interrupt handling.

4.11 Tuning the lock system

Because we had a previous experience in MP UNIXes, we decided to shoot directly for a 4-way MP-efficient for the first release. A number of guidelines were given to developers. The *collision section* of a lock is the % of the total CPU time spent under this lock in the uniprocessor case. The objective was that no collision section should be greater than 2.5%, for significant benchmarks (e.g. TPC-C, SDET, LADDIS). This can be measured with the LCA tool on a UP system. So even without any SMP HW available, we were able to determine early during the development process all the potential granularity problems and established a list of "locks to be broken". We also identified too fine-grained locks which had to be coalesced.
We believe that the scheme put in place can scale up to 8-way without major redesign, requiring only a series of measure-and-tune cycles that we will implement between the first and the second MP AIX releases.

4.12 User mode locks

Some applications and most importantly database managers implement locks in user-mode because they cannot afford the performance overhead of system provided semaphores (System V IPC) and started implementing multithreaded applications before a standardized API was available. These locks are often called latches.

On a UP AIX 3.2, these locks are implemented with a *compare_and_swap()* library routine which uses the *load_and_reserve* and *store_conditional* instructions on PowerPC machines or a system call on POWER machines, where atomic instructions are not available.

On MP machines, this scheme does not work because of the weakly ordered memory model described in the HW section. As the insertion of the fence instructions is a delicate process and moreover because these fence instructions depend on the type of PowerPC processor (601, 604 or 620) for optimal performance, we provided a simple API to hide this complexity:

_check_lock()	same semantics as *compare_and_swap()* and issues the appropriate import fence.
_clear_lock()	issues the appropriate export fence and clears the lock-word.

Programmers can then combine these to implement the appropriate type of lock for their application.

An alternative is to use the POSIX1003.1c mutexes and condition variables provided with the pthreads library.

5. Other atomic operations

The availability of *load_and_reserve* and *store_conditional* instructions enables the construction of other atomic operations. They provide better performance when compared to locks for certain operations. They usually do not need fences because no data other than the parameter of the operation is implicitly involved.

We use mainly:

_fetch_and_add	to implement counters, statistics
_fetch_and_and|or	to implement bit masks
_compare_and_swap	to implement some carefully chosen list insertion/deletions primitives (singly-linked lists)

6. Other SW issues on an SMP system

6.1 Scheduling and dispatching: affinity

The architecture being what it is, the natural tendency is to let the scheduler dispatch threads on processors from a single run-queue and according to priorities.

However, the presence of the large Level-2 caches creates some affinity between threads and processors because the working set of threads tends to accumulate in the caches [Tor92]. So if the thread is migrated to another CPU and therefore another pair of caches, a warm-up of the cache is unavoidable, with a burst of lateral misses (i.e. miss where data comes from other caches, as opposed to vertical misses where data comes from main memory). This temporarily high miss rate decreases the performance.

So we implemented an affinity scheduler with a simple algorithm to ensure that the pathlength increase in the scheduler code is minimal, so that the global outcome is positive. Each thread remembers its level of affinity with processors (actually, it is simply the last processor where the thread executed). When a processor becomes idle, the scheduler scans the runqueue and dispatches preferably a thread with affinity for it. The algorithm must however ensure fair access to CPU time. Several parameters (e.g. length and depth of the runqueue scan) were tuned to insure this.

6.2 Binding threads to processors

An API allows the programmer to explicitly bind a thread or a process to a specific processor, preventing the scheduler to migrate it when possible. Caution is recommended with this user-driven scheduling, because the scheduler generally does a better job in terms of global performance optimization. It should therefore be used only for special dedicated applications (e.g. real-time).

6.3 Managing interrupts

Masking against interrupts is done by writing some mask value into a HW register associated with the CPU. When the CPU is running at thread level, i.e with no interrupts masked, it writes a code in this register to distinguish the idle task from the normal processing. This register is used by HW to dispatch interrupts to the "least loaded processor", so all interrupts will go to one of the idle processors or, if none, to one of the processors not masked against interrupts.

As interrupt handling involves significant overhead, this "interrupt steering" improves response time. Maximum throughput is not much improved since, in that case, CPUs are seldom idle.

6.4 Clock management

Ideally, only one clock interrupt per system (as opposed to one clock interrupt per CPU) is needed on an SMP since almost all clock related actions are independent of the CPU. However, actions such as profiling a CPU i.e. determining which piece of code it is currently executing, need interruption of the CPU. So we ended up with a scheme where each CPU is interrupted at each clock tick but only one of them implements the system-wide time-related tasks.

6.5 Multi Processor communication (MPC)

SMP kernel implementations require some sort of basic inter-CPU communication. In AIX Version 4, it is called MPC.

The MPC allows a processor to send an interrupt to another processor and have it perform a specific action. We tried to maintain this layer as simple as possible. The MPC services allow:

- To register a service routine, which will be called by the target processor upon reception of an inter CPU interrupt.
- To trigger an inter CPU interrupt on a specific processor or on all the other processors (broadcast).

The MPC services are used for various functions such as funneling, timer management, dump and kernel debugger (to stop all the other processors).

7. Overall results

The results of all these technical choices can be summarized in scalability figures for various types of workloads. These figures being highly sensitive, you will have to refer to the marketing brochures of both companies to get them. Suffice it to say that from a technical standpoint we are very happy with the overall scalability.

8. How we did it

AIX 4.1 is the result of a joint development program between Bull (Grenoble, France) and IBM (Austin, TX). Bull brought its SMP experience in DPX/2 and DPS7000 and IBM its knowledge of AIX and the PowerPC architecture.

The architectural design was done by a joint team of SW architects. The work sharing was done by a joint team of technical managers. The component level design was done by the teams where each component responsibility had been assigned.

A common team in a single location did the locking of the kernel core because these parts interact so heavily that a single team was necessary. The team was composed of engineers from both companies.

The rest of the work in kernel, libraries and commands was separable and so could be dispatched between Austin and Grenoble with the help of a fast transatlantic communication link allowing us to work on the same source tree with a unique configuration management system in almost real time.

The whole project lasted all in all more than 2 years, from early Bull-IBM technical contacts to shipment. However, this involves the SMP HW development and the entirety of the AIX Version 4 development cycle. This latter decision must be drawn to the attention of those who may have heard of shorter development cycles with respect to MP-enabling a UNIX OS implementation. Instead of freezing the functionality of AIX 3.2 to make it support SMP HW and let AIX evolve functionally in parallel to meet market requirements on UP only, we commonly decided to proceed with a single development path. This by no doubt made the MP enablement more complex, but avoided the confusion and costs resulting from having 2 versions of AIX.

So it is difficult to size the SW SMP effort per se, but it lasted around 18 months (design to General Availability) and involved more than 100 people.

With a lot of good will on both sides, this cooperation worked and is still working incredibly well, probably due to our common technical culture which is a mix of UNIX background and mainframe experience.

9. References

This is not a complete bibliography on MP aspects of the UNIX kernel, because it would be too big.

[Cam91] Lock Granularity Tuning Mechanisms in SVR4/MP, Mark D. Campbell, Russ Holt, John Slice, USENIX SEDMS II, March 1991, p. 221

[Car93] Measuring Lock Performance in Multiprocessor Operating System Kernels Joseph P. CaraDonna, Noemi Paciorek, Craig E. Wills, USENIX SEDMS IV, Sept 1993, p. 37

[IBM94] PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, Prentice-Hall 1994

[Kor89] Sema: a LINT-like tool for analyzing semaphore usage in a multithreaded UNIX kernel, Joe Korty, 1989 Winter USENIX

[Pac91] Debugging Multiprocessor Operating System Kernels, N.Paciorek, S.LoVerso, A.Langerman, USENIX SEDMS II, March 1991, p. 185

[Pow91] SunOS Multi-thread Architecture, M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, 1991 Winter USENIX, p. 65

[Rit84] A Stream Input-Output System, D.Ritchie, AT&T Bell laboratories Technical Journal, Vol. 63 no. 8 (October 1984)

[Ste93] WARLOCK - A Static Data Race Analysis Tool, N.Sterling, 1993 Winter USENIX, p. 97

[Tor92] Evaluating the benefits of cache affinity scheduling in shared memory multiprocessors, J.Torellas, A.Tucker, A.Gupta Technical report: CSL-TR-92-536 (Stanford Univ.) August 1992

10. Trademarks

11. Author information

Jacques Talbot is responsible of SW Architecture in the Open Systems department at Bull Grenoble since 1989. He was previously in charge of the Bull BOS UNIX kernel development, and then project manager for the Bull DPX/2 200 UNIX platform. He graduated in 1972 from the Ecole Supérieure d'Electricité.
Address: 1, rue de Provence 38130 Echirolles FRANCE
Email: J.Talbot@frec.bull.fr

This paper has been written by Jacques Talbot from Bull. However, it is the result of the work of many people in both Bull and IBM. Special thanks to Jack O'Quin, John O'Quin, Jeff Peek from IBM and André Albot, Corrado Clementi, Jean-Jacques Guillemaud, Steve Hinde (*), Michel Sanchez from Bull.

(*) now with another company