IBM POWER5 and POWER5+ processors do not fully implement cumulative ordering.

Overview:

The following information applies only to IBM® POWER5™ and POWER5+™ processors. All IBM Systems based on IBM POWER4™, POWER6® and POWER7™ processors are unaffected.

If any of the following approaches is used, then either cumulativity is not required, or the portion of cumulativity that is not fully implemented in POWER5 or POWER5+ processors is not required.

1) Use traditional synchronization primitives such as locking or RCU (Read Copy Update) that do not require cumulativity.
2) Disable SMT.
3) Enable SMT, but use Operating System processor affinity assignment mechanisms (for example, the Linux taskset command or the IBM AIX® bindprocess command) to ensure that no more than one thread of a multithreaded program, or of a set of programs that share storage with one another, executes simultaneously on the same POWER5 or POWER5+ core.
4) Enable SMT, but use Operating System processor affinity assignment mechanisms to ensure that all threads of a given multithreaded program, or of a given set of programs that share storage with one another, are confined to execute on the same POWER5 or POWER5+ core.

If none of the preceding approaches can be used in a particular case, the software changes described below will prevent a program from depending on the aspect of cumulativity that is not fully implemented in POWER5 and POWER5+ processors.

Description and workaround

In what follows, the term "processor" is used as defined in the Power ISA Books I and II. In other words, each thread on a simultaneous multithreaded (SMT) core, such as POWER5 or POWER5+, is considered to be a "processor."

In certain rarely occurring cases, the POWER5 and POWER5+ processors do not fully implement cumulative ordering (Power ISA, Book II, Section 1.7.1) as specified for the various *Memory Barrier* instructions in the Power ISA.

Cumulative ordering, or cumulativity, is not necessary for the correct operation of many multi-processor programming primitives. In particular, if synchronization is accomplished through a single location in memory, cumulativity is not involved. The most common example of this case is critical sections (such as locking) that are synchronized using a single variable in memory that is accessed using the ***lwarx*** and ***stwcx.*** instructions. The correct operation of such critical sections is not affected. (In this

document, ***lwarx*** and ***stwcx.*** are used as representatives of *Load And Reserve* and *Store Conditional* instructions. Any other pair of *Load And Reserve* and *Store Conditional* instructions could have been used instead.)

In addition, cumulativity is a property involving a minimum of three processors and synchronization through a minimum of two different memory locations. In cases in which a *Memory Barrier* instruction is used to provide ordering only for storage accesses performed by a given processor relative to a given other processor, cumulativity is not involved.

In summary, cumulativity, in certain rare cases, is not fully implemented in POWER5 or POWER5+ processors. However, ordering of storage accesses between a given pair of processors, and many multiprocessor programming primitives, most notably critical sections, are not affected by cumulativity and operate correctly on POWER5 and POWER5+ processors.

Cumulativity description

Cumulativity involves controlling the ordering of storage accesses across three or more processors. It is defined in terms of two sets of storage accesses, called "set A" and "set B," which are defined relative to a given memory barrier.

Memory barriers have a cumulative and a non-cumulative ordering component, both of which are defined in terms of sets A and B.

The descriptions in the remainder of this section are provided for illustration only. Please refer to the Power ISA documents, especially Book II Section 1.7.1, for precise descriptions of the cumulative and non-cumulative ordering associated with memory barriers

For non-cumulative ordering, set A consists of the storage accesses that are caused by instructions that *precede* the instruction that creates the memory barrier, and set B consists of the storage accesses that are caused by instructions that *follow* the instruction that creates the memory barrier. The non-cumulative ordering done by the memory barrier ensures that all accesses in set A are performed, with respect to any given processor, before any access in set B is performed with respect to that given processor. Non-cumulative ordering orders only accesses that are performed *by the processor that creates the memory barrier*.

Cumulative ordering adds, to sets A and B, accesses that are performed *by processors other than the processor that creates the memory barrier*.

Added to set A are the storage accesses by other processors that are performed, with respect to the processor that creates the cumulative memory barrier, *before* the memory barrier is created.

Added to set B are the storage accesses by other processors that are performed *after* that other processor reads a value that was stored by a store that is already in set B. This definition is recursive. Initially set B consists only of accesses caused by instructions that follow the instruction that creates the cumulative memory barrier. (These accesses were performed by the processor that creates the cumulative memory barrier.) Any store by another processor that is then added to set B can, if observed by another processor, cause additional accesses to be added to set B.

These augmented sets A and B are ordered, by the (cumulative) memory barrier, as described above. That is, all accesses in set A are performed, with respect to any given processor, *before* any access in set B is performed with respect to that given processor.

The non-cumulative behavior of memory barriers is fully implemented in POWER5 AND POWER5+ processors; only certain aspects of cumulativity are not fully implemented.

Cumulativity can be considered to consist of two properties. This discussion refers to these properties as "A-cumulativity" and "B-cumulativity."

A-cumulativity is the property that ensures that storage accesses, *by other processors*, that are in set A are performed, with respect to any given processor, before storage accesses that are in set B are performed with respect to that given processor.

A-cumulativity is illustrated in Example 1 of the second Programming Note in Section 1.7.1 of Book II. In that example, the store to X by Processor A is in the set A of the memory barrier created by Processor B (by A-cumulativity), and Processor B's store to Y is in the set B. Therefore the fact that Processor C's load from Y returns the new value means that Processor C's load from X must return the value written by Processor A.

Similarly, B-cumulativity is the property that ensures that storage accesses, *by other processors*, that are in set B are performed, with respect to any given processor, after storage accesses that are in set A have been performed with respect to that given processor.

B cumulativity is illustrated in Example 2 of the second Programming Note in Section 1.7.1 of Book II. In that example, the store to X by processor A is in the set A of the memory barrier created by Processor A, and the store to Y is in the set B. Because Processor B stores to Z after reading the new value of Y, Processor B's store to Z is also in set B (by B-cumulativity). Therefore the fact that Processor C's load from Z returns the new value means that Processor C's load from X must return the value written by Processor A.

These examples illustrate that a minimum of three processors and a minimum of two memory locations are required before a program needs cumulativity. If only two processors or only one memory location are involved, then cumulative ordering is not needed.

POWER5 AND POWER5+ workarounds

If none of the approaches listed above can be used in a particular case, the following software changes will prevent a program from depending on the aspect of cumulativity that is not fully implemented in POWER5 AND POWER5+ processors. In the following procedures, "***hwsync***" is used as shorthand for "***sync*** with L = 0," where L is the operand of the *Synchronize* instruction.

1) Replace every ***lwsync*** and ***eieio*** instruction, that is being used to ensure B-cumulative ordering of accesses to cacheable storage, with an ***hwsync*** instruction. Only ***lwsync*** and ***eieio*** instructions that are being used to ensure B-cumulative ordering, as described above, need be replaced.
2) Replace every *Load* instruction, that is being used to establish A-cumulative ordering of accesses to cacheable storage, with a ***lwarx*** or ***ldarx*** instruction (or a sequence of ***lwarx*** or ***ldarx*** instructions, for unaligned loads or loads with a longer target location).

In the event that it is not practical to analyze the program sufficiently to apply the two replacements described above, the following software changes can be used instead.

1) Replace every ***lwsync*** and ***eieio*** instruction that is being used to order accesses to cacheable storage with an ***hwsync*** instruction.
2) Replace every ***hwsync*** instruction (including those created in step 1) and every ***ptesync*** instruction with the following code.

```
        # r3 contains the address of a scratch location
        # in cacheable storage

                li      r0,8            # prepare to execute loop 8 times
                mtctr   r0
        loop:   dcbf    0, r3           # flush scratch location from the data cache
                sync/ptesync            # this matches the instruction being replaced
                                        # "sync" is sync with L=0 ("hwsync")
                std     r0, 0(r3)       # store to scratch location
                                        # (value stored doesn't matter)
                bdnz loop               # execute loop 8 times
```