IBM POWER6 reliability

M. J. Mack
W. M. Sauer
S. B. Swaney
B. G. Mealey

This paper describes the state-of-the art reliability features of the IBM POWER6™ microprocessor. The POWER6 microprocessor includes a high degree of detection of soft and hard errors in both dataflow and control logic, as well as a feature—instruction retry recovery (IRR)—usually available only on mainframe systems. IRR provides full hardware error recovery of those registers that are defined by the instruction set architecture. This is accomplished by taking a checkpoint of the defined state for both of the core threads and recovering the machine state back to a known good point. To allow changing memory accessibility without using different page table entries, the POWER6 microprocessor implements virtual page class keys, a new architectural extension that enables the OS (operating system) to manage eight classes of memory with efficiently modifiable access authority for each class. With this feature, malfunctioning kernel extensions can be prevented from destroying OS data that may, in turn, bring an OS down.

Introduction

As both software and hardware become more complex in server designs, increased investment in reliability has become necessary to keep systems running despite intermittent hardware error conditions or badly behaving or malicious software. The design of the IBM POWER6* microprocessor has added a higher level of tolerance for these conditions through several features. One is the virtual page class key protection architecture extensions that aid OS software to effectively manage memory accesses across applications, OS kernels, and kernel extensions; another is an extensive hardware investment in error detection and recovery through the use of instruction retry recovery (IRR).

This paper first presents the virtual page class key protection architecture and its initial implementation in the POWER6 processor design. The paper then describes the hardware facilities implemented in the POWER6 processor to tolerate soft and hard errors occurring in the system.

Virtual storage protection

IBM Power Architecture* technology [1] provides a mechanism for virtual storage protection, which is based on the MSR(PR) (machine-state register problem-state) bit; the problem-state storage key (KP) and the supervisor-state storage key (KS) bits in the segment look-aside buffer (SLB) entry; and the page protection (PP) bits in the page table entry (PTE). This mechanism controls load- and store-type accesses when address translation is enabled. There is an additional mechanism to prohibit instruction execution from a storage page.

The mechanism used in the POWER5* processor requires a very small amount of resources and is, therefore, very efficient. However, its function is limited because there is no easy way to maintain different access rights for software that uses common SLB entries and a common page table. For example, the IBM AIX* OS kernel and kernel extensions use the same page table and, therefore, have the same access rights. Most kernel data is contained in a flat address space that includes many memory classes, such as cached files, network buffers, application data, and core kernel data. Kernel-mode programming errors can result in the corruption of any memory class. Applications face similar problems with protecting their large address spaces.

Changing access rights in this context would require using a different page table or at least invalidating sensitive PTEs, both of which are costly actions, since translation look-aside buffer (TLB) entries would have to be invalidated, and new translations would have to be loaded via page fault interrupts and subsequent lookups of PTEs that reside in main memory.

©**Copyright** 2007 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/07/$5.00 © 2007 IBM

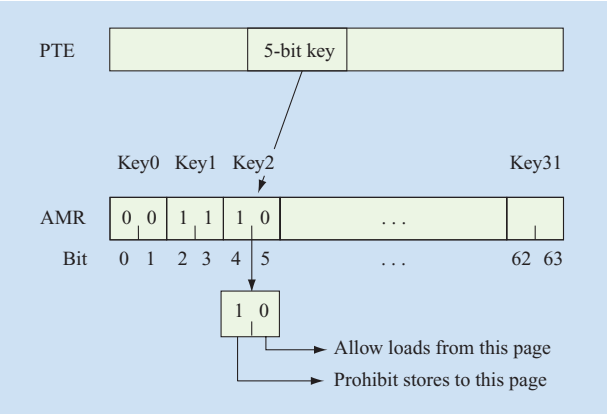

Figure 1

Page table entry (PTE) and access mask register (AMR).

Table 1 Access mask encoding for virtual storage key k.

AMR value	*Effect on loads and stores*
AMR_{2k}, $AMR_{2k} + 1 = 0, 0$	Store access permitted; load access permitted
AMR_{2k}, $AMR_{2k} + 1 = 0, 1$	Store access permitted; load access not permitted
AMR_{2k}, $AMR_{2k} + 1 = 1, 0$	Store access not permitted; load access permitted
AMR_{2k}, $AMR_{2k} + 1 = 1, 1$	Store access not permitted; load access not permitted

The virtual page class key protection extension to the Power Architecture instruction set is a lightweight mechanism that allows altering access rights to storage without changing the page table or invalidating cached translation entries. This mechanism defines access rights on the basis of a page granularity for flexibility in assigning data pages to memory classes with common access rights.

As indicated in **Figure 1**, the mechanism operates as follows: The PTE contains a new 5-bit virtual storage key (the drawing simplifies the real situation, because the key is actually split in two parts—one 2-bit and one 3-bit portion to maintain compatibility with the existing architecture). The virtual storage key is used to index the access mask register (AMR), which is a new special-purpose register (SPR). The AMR consists of 32 2-bit access keys (access masks). One bit controls write access, and the other controls read access, allowing the combinations shown in **Table 1**.

The AMR is a privileged register that can be set by the OS to any desired value. Changing the value is fast and does not require a PTE change or the invalidation of cached translation entries.

Software uses the AMR to limit access to memory classes. Programmers determine the memory classes that a software component was designed to access. The 5-bit storage key in the PTE defines membership in one of 32 memory classes for the virtual page. A *memory class set* (the collection of virtual memory pages with the same storage key, and thus in the same memory class) is maintained in the AMR to protect against invalid memory references. There are several examples in the AIX kernel. For example, one AIX kernel memory class is application data. Under the AIX OS, only a small amount of code requires direct access to application data, so when software components that require application data access are entered, they enable the memory classes in the AMR. Most code in the AIX kernel runs without an application data class, i.e., without a key, and is prevented from making accidental references to user-mode data. Another example is that the core kernel protects its critical data with a memory class that is not in the AMR when running kernel extensions.

In the AIX kernel, the AMR is normally loaded on module entry and restored when a module returns. This will not protect against malicious kernel extensions running in privileged mode, since they can manipulate the AMR and gain access to protected pages. Kernel extensions are generally much better controlled and are trusted. Virtual storage keys are used exclusively to improve OS reliability in this environment.

Applications can also use the AMR to limit memory accessibility. The kernel provides fast system calls to allow applications to alter their user-mode AMR. Applications can use keys to define user-mode memory classes. For example, plans for the IBM DB2* relational database management software system call for limiting accessibility to its buffer pool when it makes callouts to user-defined functions (customer code dynamically loaded into the database). Other independent software vendors are interested in segregating memory into classes similar to the AIX kernel.

The architecture allows up to 32 virtual storage keys. Eight virtual storage keys are implemented in POWER6 processor-based hardware. The AIX 5.3 OS with the POWER6 processor update supports the application use of keys. It is planned to implement kernel usage in AIX 5.4 OS.

Hardware recovery

Because of particle emissions from the decay of radioactive atoms and from cosmic rays, soft errors occur in microprocessors [2], SRAM (static RAM), register

files, and even in latches. At current design lithographies (65-nm technology is used for the POWER6 processor), soft-error rates are increasing, and attempts to harden latches to be impervious to particle emissions are less successful because of their small size and the need to avoid having them be excessively large structures.

Strategies are required to detect such errors, correct them if possible, and if not, recover from them. For example, in the SPARC64** processor, which has an out-of-order execution design described in [3], instruction results that report errors and any subsequent instructions are canceled before being committed. The youngest instruction not to be committed is retried, i.e., reissued singly, and if successful, normal superscalar operation continues. To isolate these occurrences, error-detection circuits in both the execution units and the datapaths were added to the SPARC64 architecture.

In IBM zSeries* processors [4], to achieve robust instruction retry, two copies of each execution unit are included, and their results are compared at the completion of every instruction. If they differ, the result is not committed to the recovery unit (RU), which contains the facilities defined by the instruction set architecture (ISA) and is protected from single- and double-bit errors by error-correction code (ECC). The RU sends its data to the copies in the execution units and retries the instruction.

Previous generations of POWER* processors were able to tolerate failures in large arrays, such as the level 1 (L1) and level 2 (L2) caches, by using inline correction techniques, but failures in latches or combinatorial logic were rare enough that no significant investment was made to tolerate them or even directly detect them. Thus, failures in the nonarray logic typically resulted in a system checkstop or system hang. The POWER6 processor has a much greater investment in detecting and tolerating failures in both arrays and logic.

In the POWER4* and POWER5 processor designs, error recovery was limited to using the machine-check interrupt handler to recover from correctable and uncorrectable errors (UEs) in SRAMs. Error checkers in the control flow and the dataflow produced checkstop conditions and recorded occurrences for fault isolation [5].

The POWER6 processor design added error-detection logic to all dataflow and most control flow, achieving a much higher degree of soft- and hard-error recovery than previous POWER processor designs. In contrast to the zSeries processor design approach, the area consumed by the recovery function is far less than that required to duplicate each execution unit in the design, while both designs contain an RU to hold the ECC-hardened copy of facilities defined by the ISA.

The POWER6 processor approach—which uses a separate RU rather than embedding the function in the completion logic, as is the case in the SPARC64 design—allows for a lesser degree of coupling between error events and the halting of checkpointing,[1] and it also decentralizes their synchronization between functional execution units.

Checkpointing and recovery

The POWER6 processor implements IRR, which is the same concept that has been used in zSeries processors for several generations. The fundamental concept of IRR is to maintain an architectural checkpoint on hardware instruction boundaries. The checkpoint can be restored in the event of an error so that processing can be retried and, hopefully, resumed from the last instruction checkpoint. Instruction checkpointing has dependencies on the logic throughout the processor:

- A means of preserving the entire state of the processor in a hardened checkpoint (protected by ECC).
- Protecting the integrity of the checkpoint with robust error detection throughout the processor.
- A means of resetting noncheckpointed logic to attempt to remove the error.
- A means of restoring the checkpoint.

The register checkpoint is implemented using an RU. All registers for both threads are captured in the RU with the checkpoint maintained on instruction group boundaries. The RU has a pipeline for capturing register results so that the checkpointing lags execution, which is important so that errors can be detected in time to block bad results from being included in the checkpoint.

Error detection throughout the processor core is essential to protecting the checkpoint. Errors must be reported from all areas of the processor, and logically ORed together to block checkpoint updates. Errors that are detected too late to be reported in time to block checkpointing must be escalated to a processor checkstop. Errors that affect the restoration of the checkpoint must also be escalated to a processor checkstop, but as long as the checkpoint is intact, restoration on an alternate processor may be possible.

Recovery is performed on a processor-core basis, not on an instruction or thread basis. Reported errors do not have to be associated with an instruction boundary as long as they are reported early enough to block checkpointing. IRR may back up to a checkpoint that occurred earlier than the instruction that encountered the error. Both threads will be restored from the most recent

[1] *Checkpointing* is the action of capturing a known good state (i.e., checkpoint) of the registers defined by the ISA and sorting it for possible future use. While many sequential checkpoints may be retained as part of a checkpointing process, the POWER6 processor maintains only one checkpoint at a time; a newly created checkpoint overwrites the previous one.

checkpoint regardless of the error that was detected or the thread on which it occurred.

To make IRR more effective for functional errors, the execution flow through the processor core is simplified (i.e., single dispatch, nonpipelined) for some short amount of time after recovery. Recovery is considered successful when enough instructions execute successfully for the checkpoint to advance beyond the simplified execution mode and full-speed execution is resumed without error.

It is desirable to have a single recovery mechanism regardless of the error that is detected. However, certain complexities and environments require special consideration. These include the following:

- L1 cache and its associated directories and look-aside buffers.
- The timing facility, which includes the timebase (TB), decrementer (DEC), hypervisor decrementer (HDEC), processor utilization of resources register (PURR), and scaled PURR (SPURR).
- Cache-inhibited loads.
- Store conditional instructions (`stwcx` and `stdcx`), which are used along with the load and reserve instructions to permit atomic update of a storage location.
- Memory barrier instructions, such as synchronize.
- TLB management instructions, such as TLB invalidate entry (`tlbie`), that affect multiple processors in a system.
- Cache and memory UEs.
- Data aborts.
- The segment look-aside buffer (SLB).
- Noncheckpointed SPRs.
- Store-related errors.
- Thermal events.
- System-level firmware support.

Many of these special considerations require host firmware support. The reporting of recovery and status to the hypervisor is handled by the new hypervisor maintenance interrupt (HMI).

This IRR strategy is very effective for soft errors (transient errors induced by radioactive particles striking latches), but not for permanent errors. Permanent errors that do not allow the processor to make forward progress are escalated to a processor checkstop. However, a processor checkstop does not take down the entire system, but rather invokes higher-level system recovery.

RU checkpoint

Instructions are checkpointed on group boundaries within each thread. At the time of dispatch, for each thread, a group tag (GTag) is sent along with the instructions to denote the age of the group relative to each instruction tag (ITag) and is used to determine when a group can be checkpointed. A group can be checkpointed when the next-to-complete ITag is equal to or greater than the GTag. When a group is partially flushed because of branch misprediction or load/store reject, or when an instruction causes an exception, a new GTag must be recalculated and saved by the RU. This information is used to allow the RU to partially checkpoint the original dispatch group while discarding the data of the flushed instructions.

The fixed-point unit (FXU) and load/store unit (LSU) data must wait at the RU for the floating-point unit (FPU) or vector multimedia extension (VMX) unit instructions in the same dispatch group to be completed before the whole dispatch group can be checkpointed. This is because the FPU and VMX instructions take more cycles to execute than fixed-point, load/store, or branch instructions. The FPU and VMX instruction results are kept in separate queues from the fixed-point instruction results. At dispatch time, information from the dispatch group about the expected number of results of FPU instructions and whether VMX instruction results are expected is sent, along with the dispatching instructions, to the checkpoint queues in the RU. The group can be checkpointed only when all FPU or VMX data for that group is available. When a group is checkpointed, the RU indicates to the store queue that any stores from that group can be released to the L2 cache.

Recoverable errors are ORed together to block further checkpointing. In order for an error to be recoverable, it must be reported to the RU in time to block the checkpointing of any instructions possibly affected by the error. For some errors, the latency to report them through the fault isolation register (FIR) structure would make them unrecoverable, so they are sent directly from each functional unit to the RU. After the recovery state machine completes the reset to clear the error, the RU restores the checkpointed register values to the functional units. The RU has separate buses to the FXU, FPU, and VMX for sending the checkpointed register values, which the functional units wrap back onto the normal functional update buses to restore the values to the working copies.

Recovery sequence

When an error or minirefresh (see the following section) request is active on the chip, the recovery state machine sequences through the necessary operations:

1. First, the machine waits a programmable amount of time to determine whether the error must be escalated to a checkstop, as the first reporting of an error may not indicate that a checkstop is required.

A cascading error or an error that arrives after the first report may require a checkstop.

2. A lookup is done on the GTag in the RU to resolve its mapping to the instruction address that will be executed next upon completion of IRR. The result is stored in the RU for subsequent refresh.
3. The L2 is instructed to be quiescent, as the core cannot respond to snoop requests and will not generate further traffic.
4. A fence signal is raised to the L2 to indicate that it should ignore all signals coming from the core, as the process of IRR may generate random signal transitions on its interface.
5. All SRAMs in the core go through array built-in self test (ABIST) to clear any existing state information. Because the L1 cache is a store-through design, no castouts to the L2 are required. In the case of a minirefresh, this step is omitted.
6. All registers defined by the ISA are refreshed to the core from the checkpointed state in the RU. Fixed-point, floating-point, and vector registers are routed through separate buses to their respective units in a defined order. All SPRs are routed through the FXU via the normal functional path used for a move to SPR instructions.
7. A hardware reset signal is sent to all units to clear any required state that could be corrupted or hung by the error event. The state machine then waits for a signal that ABIST has completed, as this operation requires much more time than the register refresh and hardware reset functions.
8. All FIR bits are reset.
9. All registers are refreshed again, as in step 6, to ensure that all existent errors are reset and that the ABIST and hardware reset operations did not cause any error conditions in those registers.
10. The fence signal to the L2 cache is dropped, and the L1–L2 interface now operates normally.
11. Instruction execution resumes from the restored checkpoint state in a reduced execution mode, e.g., single-instruction dispatch, nonoverlapped FPU execution. This state continues until a hardware programmable number of instructions have completed successfully in order to ensure that the instruction that caused the original error has been executed. For IRR, the HMI handler is invoked first to log the event and perform any necessary software assistance, such as SLB restoration. Thus, the number of instructions executed in this state includes the HMI handler itself plus a sufficient number of instructions to ensure that the error-causing instruction is repeated.
12. Once the successful instruction completion threshold is exceeded, normal operation resumes.

Minirefresh

The minirefresh function is used to implement hardware work-arounds during early hardware bringup and to aid the performance of two functions. Logic exists in the core to detect certain conditions based on debug facilities that can cause a minirefresh. The minirefresh allows the offending instruction to be executed in a reduced execution mode, which will avoid the problematic case.

Additionally, because of timing requirements, early hardware implementations required trap instructions to be dispatched in a single instruction group to resolve the trap condition. This posed a performance problem for certain code streams, as the incidence of taken trap instructions is relatively low compared to the number of trap instructions inserted in the stream.

To assist in this case, the FXU signals a minirefresh request when a taken trap is detected in a multiple instruction group. The minirefresh request blocks the checkpointing of the dispatch group that contains the trap instruction as well as all subsequent dispatch groups. A minirefresh that does not include the ABIST step is executed, and execution resumes prior to the trap instruction in single-instruction dispatch mode, thus avoiding the timing issue arising from multiple instruction groups containing traps.

Also, precise floating-point exception mode is implemented in the POWER6 processor by operating in single-instruction dispatch, non-floating-point pipelined mode. For certain applications that require precise interrupt execution, this penalty is too high. A fast floating-point precise mode was added that uses the minirefresh function.

When in fast floating-point precise mode, any floating-point exception signals a minirefresh request to the RU and stops the checkpointing of data associated with the excepting instruction. As with the trap function, a minirefresh is executed, and the offending instruction reexecutes in single-instruction dispatch mode with floating-point pipelining disabled, allowing the interrupt to be reported precisely.

For both operations, a minirefresh requires of the order of 300 processor cycles, so the penalty for this function is low as long as the incidence of taken traps and floating-point exceptions is low relative to the instruction stream. Thus, it provides a performance boost compared with not using these mechanisms.

Special cases for recovery

L1 cache and associated facilities

The IRR mechanism is effective for soft failures, but hard failures that prevent the processor from making forward

progress are not recoverable with IRR. However, because of the high number of circuits in the POWER6 processor, large SRAMs, such as the L1 cache (and its directory), support delete mechanisms to work around hard failures. The L1 cache implements the delete on a set boundary. (A *cache set* is one of *n* locations where a cache line may be placed when it is loaded from the next level in the memory hierarchy. Each cache line in the real memory space of the processor can be loaded into any one of the *n* sets. The POWER6 processor L1 instruction cache contains four sets, and the L1 data cache contains eight sets.)

Because healthy SRAMs will experience soft errors due to normal ambient radiation, a thresholding mechanism is implemented to distinguish hard errors from soft errors in order to invoke a set delete. When an L1 cache (or directory) parity error is detected, the set identification number is trapped, and a bit is set to indicate that an error was detected. If a new error is then detected and the error bit is already active, and if the set identification matches the trapped set identification from the prior error, then it is assumed to be a hard error and the corresponding set is deleted (marked as unusable). Periodically (every few seconds), the error indication bit is cleared so that a newly detected error will be assumed to be an independent soft error. Because it may be several cycles from the reporting of a parity error until cache accesses cease, the threshold is set so that it does not overindicate as a result of multiple parity errors from the same event. When a parity error is detected, indication of a new parity error is blocked until the IRR process.

The set delete lines for associated arrays (data, directory, and set predict) are activated together. When a set is deleted, the LSU or instruction fetch unit (IFU) must indicate that a set was deleted, which set it was, and which array contained the error that caused the deletion. Errors from separate arrays set separate FIR bits. The delete indication and set identification are explicitly reported to the core pervasives unit and held in a recovery status register.

Because a hard error must be detected twice before the set delete is invoked, it is possible to detect the error a second time without forward progress having been made after IRR. The IRR logic makes this possible by implementing a programmable no-forward-progress threshold. Each time an IRR is executed but no forward progress of instruction execution is achieved, a counter is incremented. IRR will be repeated until forward progress is achieved or the counter exceeds the programmable threshold value, resulting in a processor checkstop.

Timing facility (TB, DECR/HDECR, PURR/SPURR)

A dedicated time-of-day (TOD) oscillator, which is not subject to spread-spectrum or dynamic frequency variation, is used to step the TB and DEC registers. Thus, these registers are not associated solely with processor frequency; they are able to keep real time regardless of processor frequency. Because the timing facility registers are updated in real time, they are not tied to instruction boundaries, as are registers checkpointed in the RU. Thus, these registers cannot be recovered by IRR; they require assistance from firmware.

Each processor chip has a TOD register that is synchronized peer to peer with all other processor chips in the system. The chip TOD has an interface to each processor core TB, enabling all processor cores to have synchronized TB registers. The initial time setting requires a firmware-driven sequence that uses the move-to-SPR instruction (`mtspr`) to the TB and a new timer facilities management register (TFMR) SPR.

The same TOD value exists across all chips, and this redundancy is used for restoring individual TB values in the event of failures. DEC/HDEC or PURR/SPURR failures, if corrupted, must be restored by host firmware. Errors in the timer facilities are reported to the hypervisor via HMI. A bit in the hypervisor maintenance exception register (HMER) indicates a timing facility error, and bits in the TFMR provide more information about the state and health of the various timer facilities.

Cache-inhibited loads

Some cache-inhibited (CI) loads cause coherent storage locations to be modified, so they have the same affect as stores. However, because a CI load cannot complete until after the data returns and is written back to the general-purpose register, the store is done prior to the checkpoint of the CI load. This creates a special case for recovery, as IRR cannot back up to a checkpoint prior to the CI load and retry it, because the storage has already been modified.

The RU always allows a CI load to checkpoint, even if checkpointing is blocked because of a reported error. Dispatch restrictions on CI loads ensure that a CI load will be the only instruction in the group for that thread, so only the CI load is allowed to checkpoint past an error and no other instructions.

Therefore, in the presence of an error somewhere in the core, the dispatch and LSU dataflow logic still operate correctly in order to complete a CI load and advance the checkpoint. Thus, errors in this logic while a CI load is in progress escalate to a processor checkstop (with unique FIR bits to identify this window).

The mechanism of requiring CI loads to checkpoint after an error is detected reduces the recoverability of soft errors in the dispatch and LSU dataflow logic by approximately 2%, as it is expected that CI loads will be outstanding approximately 2% of the time. It still allows full SRAM soft-error recovery.

STCX

Store conditional doubleword indexed (`stdcx`) and store conditional word indexed (`stwcx`) instructions are collectively referred to as *STCX*. Along with the load and reserve instructions, STCX is used to perform atomic update operations. STCX is similar to CI load in that it modifies coherent memory prior to being checkpointed using the same approach as for CI load. The only difference is that STCX is held in execution until completion, so it does not need to be redispatched when the data for the condition register (CR) and the fixed-point exception register (XER) is available.

STCX is the only instruction in a dispatch group for a thread. The LSU informs the RU when a STCX is executing on a per-thread basis. If an error is detected while a STCX is in progress for a thread, the checkpoint for that thread must be allowed to advance to the point that includes the STCX.

Therefore, in the presence of an error somewhere in the core, the logic that is updated by the STCX must still operate correctly in order to complete the instruction and advance the checkpoint. Thus, errors in this logic while a STCX is in progress must escalate to a processor checkstop.

Also, similar to CI loads, if the L2 detects a normally recoverable interface error on a STCX and squashes the STCX, it indicates to the core that the instruction was canceled. This is to prevent the core from hanging while it waits for an attempted recovery to complete.

SYNC (HWSync), PTESync, and TLBIE (TLBI-Remote)

The hardware barrier/TLB invalidate entry and heavy-weight barrier (HWBarrier) instructions use a `barrier_done` flag to ensure that only a single HWBarrier is outstanding to the storage interface. These operations must be completed, i.e., the `barrier_done` pulsed back to the core before the IRR can proceed.

HWBarrier instructions are always the only instructions in a dispatch group for a thread. The LSU does not send a barrier operation to the memory subsystem until the RU advances the GTag past the instruction, i.e., it is checkpointed. If an error is detected while an HWBarrier is in progress (released to storage) for a thread, IRR cannot proceed until `barrier_done` is received.

Thus, in the presence of an error somewhere in the processor core, the `barrier_done` logic must still operate correctly in order to complete the HWBarrier. Thus, errors in this logic must escalate to a processor checkstop.

Cache and memory uncorrectable errors

UEs may originate from memory, the cache hierarchy, or interface failures. The IBM Power Architecture includes a synchronous UE machine-check interrupt function. The processor hardware is responsible for trapping the failing storage address for the hypervisor machine-check handler, and the hypervisor is responsible for containing the damage to the affected workload and utilizing available OS recovery. For instruction fetch UEs, the hardware traps the effective address in the machine status save and restore register 0 (SRR0) and the reason in the machine status save and restore register 1 (SRR1). For data fetches, the hardware holds the effective address in the data address register (DAR) and the reason in the data storage interrupt status register (DSISR).

UEs are often quite damaging because of the limitations of the machine-check handler, depending on the address affected. Some transient UEs from the cache hierarchy have been observed in POWER processor-based systems, which suggests a failure mode in the memory subsystem, where a transient event affects persistent state, causing multiple UEs from main memory. This experience warranted an investment in improved UE handling.

For L2 UEs, the objective is to allow the L2 purge and delete (implemented in hardware) to eliminate UEs in unmodified data. The processor goes through normal IRR on the first occurrence of a UE on the chance that it will be removed with the purge and delete. While the processor is in IRR, the L2 purges the line containing the UE and deletes the line. Unmodified data is simply invalidated, and modified data will cause a special UE (SUE) to be cast out to memory. If the UE was in unchanged data, it should disappear altogether, and if the UE was in modified data, it will return (as a SUE) when refetched from memory after IRR, and the processor will take a machine-check interrupt.

In order to avoid the latency for inline ECC correction, the processor nest interface implements a level of retry below IRR. *Non-safe mode* implies that inline ECC correction is bypassed, while *safe mode* implies that inline ECC correction is enabled and UEs will be stamped as SUEs to the processor. When a UE is encountered in non-safe mode, the load instruction is rejected by the LSU and the data is re-sent in safe mode by the L2 cache. The instruction will be rejected by the LSU until the data arrives without error from the L2 after passing through inline ECC correction. If the data still contains a UE after ECC correction, IRR is entered to correct the error.

SLB

Because of the size of the SLB, it is prohibitively expensive in terms of area and power to maintain a separate SLB checkpoint. Therefore, SLB entries are backed up in a shadow memory buffer that the OS can use to restore the SLB contents, if required.

The SLB is preserved (not reset) during IRR, but some error conditions may compromise its integrity. These include parity or multihit errors and update windows in which the SLB was updated but the instruction that caused the update did not reach the checkpoint before an unrelated error was detected. The integrity of the SLB is indicated to the hypervisor via the HMI or HMER, or via a machine check, and the hypervisor notifies the OS to restore the SLB. A hardware failure in the SLB will normally cause a parity error, but it could also cause a multihit error. A software bug that causes duplicate entries will cause a multihit error, but because a multihit error ORs the data from the multiple entries, it may or may not also indicate a parity error. An SLB update window error is due to an unrelated error, not specific to the SLB, that hit at a bad time. In general, it is assumed that a multihit error is a software bug, and a parity error without a multihit error is a hardware failure. Update window errors, i.e., errors that occur in the processor while an SLB entry is being updated, are due to hardware failures outside the SLB.

As SLB restoration or recovery is not guaranteed (particularly for software bugs), information is provided to the hypervisor about the cause of the SLB corruption so that it can isolate it to a single thread. The SLB does not get reset by hardware during recovery, even when it is corrupted.

The first step is to detect and report the three possible conditions for each thread: parity errors, multihit errors, and update window errors. The LSU reports parity and multihit errors, and the RU reports update window errors for each thread. Because the hypervisor could encounter SLB parity errors during a save and restore operation, which is a window not enabled for HMI, a machine check is required to break the program flow and circumvent the error. Since this specific case of a parity error requires a machine check, a machine-check interrupt is used to report all parity and multihit errors (all SLB errors detected by the LSU). The RU-detected window error does not persist (like a parity error), so it will not prevent the hypervisor from making forward progress after recovery; thus, an HMI is adequate to support them.

Noncheckpointed SPRs

Noncheckpointed SPRs have a window similar to the SLB-modified window in which an SPR is modified, but the `mtspr` instruction did not checkpoint because of an error condition or a minirefresh trigger. The hardware will detect the SPR-modified window and report it to host firmware via HMI very similarly to how the SLB-modified window is detected. The SPR-modified condition is indicated in the HMER.

The SPR-modified window is almost identical to the SLB-modified window and requires no special handling for successful IRR, except to clear the SPR-modified bit in the HMER. The process to handle errors that occur during an SPR-modified window consists of the following steps:

- `mtspr` instructions are not to be dispatched until the RU write queue is empty, meaning the only active instruction is the `mtspr`.
- Asynchronous machine checks are not blocked.
- HMI is blocked until the core is not in hypervisor state or is not making forward progress after IRR in hypervisor state, which guarantees that if the hypervisor performed an `mtspr`, it will be retried before taking the HMI.
- Asynchronous interrupts (non-HMI, non-machine check) are blocked until forward progress is achieved in the same hypervisor state as the one in which the error was encountered. If the original error occurred while in hypervisor mode, then waiting for the `mtspr` to be reexecuted before taking the HMI will also wait to take any other asynchronous interrupt (non-machine check). If not in hypervisor mode at the time of the error, the core will take the HMI but will block other asynchronous interrupts until the `mtspr` is reexecuted in nonhypervisor mode after the HMI.

This guarantees that if there is no asynchronous machine check, then the `mtspr` will be reexecuted prior to any further access to the SPR.

Because the `mtspr` executes by itself, there are no synchronous machine-check conditions that could cause the SPR-modified window to be hit. The exposure to asynchronous machine-check conditions is also negligibly small, though not impossible. Therefore, the machine-check handler also checks for the SPR-modified bit in the HMER. Because the machine-check handler checks for the bit, the HMI interrupt handler is responsible to clear the bit.

If an asynchronous machine check does occur during the SPR-modified window, the hypervisor will checkstop the processor. Assuming the machine check was survivable in the first place, the machine-check handler should indicate "hypervisor backed up" prior to the checkstop, so that processor sparing may still be attempted. A processor could also checkstop because of a permanent fault that hits the SPR-modified window.

The registers affected by the SPR-modified window control debug and performance-monitoring functions, which are not preserved when a processor-sparing action occurs. The performance monitor must be invalidated, and the user notified to reinitialize the desired setup. Typically, hardware debug controls are the same on every processor, so they are not actually lost. The user of

software debug functions should be aware that processor sparing occurred and that it is necessary to reinitialize the debug environment.

Store errors

Storage is part of the processor checkpoint. The L1 cache is store-through, so in general, storage is checkpointed coherently outside the processor. However, for some period of time, storage updates reside in buffers only in the processor. Errors in the store buffers or on store interfaces prevent the changed data from reaching the coherent storage checkpoint. Because the store buffers and interfaces make up a substantial number of circuits, errors in this logic must be recoverable, or at least contained as much as possible, to minimize damage to the system.

The hardware must prevent storage errors from affecting other processors, which means not allowing storage errors to propagate to storage owned by other processors. Any store-related error should prevent the store from being written to the L2, so that the storage corruption is confined to a dropped store. The bounds of the storage corruption then depend on whether the store was done by the hypervisor, the OS, or a user program. If the store was done by a user program, the OS may be able to recover from a software checkpoint. If the store was done by the OS, the hypervisor may be able to contain the damage to a single partition. If the store was done by the hypervisor, it is likely that the entire system will have to be checkstopped.

The hardware must detect when it has dropped a store and differentiate whether it was a store by the partition or by the hypervisor. The core includes the hypervisor state bit with each store command, and the nest stores it through the life of the request. The L2 cache implements separate FIR bits for hypervisor and nonhypervisor state stores dropped, for each core. (These FIR bits are independent of address errors, data errors, and command errors, so that the hypervisor or nonhypervisor versions of all types are not needed.) The FIR that indicates the error will always be on the same chip as the core that issued the dropped store.

System-level firmware support

Functions such as event logging, setting thresholds, Elastic Interface bit steering, L2 and L3 cache-line deletes, and frequency and voltage adjustments or thermal management require firmware help. Hardware and host firmware are used for any functions that have real-time requirements.

All successful soft-error recoveries by the POWER6 core are logged. This includes capturing trace-array information. (The trace array is frozen at the point of the recovery. The trace is reenabled after the trace data has been captured for a log.)

A recovery may appear successful to the hardware, but it is possible that the core is not making progress from the partition viewpoint (e.g., there may be a hardware defect in the FPU that causes a recurring error). The firmware may not be able to react quickly enough to keep up with the recovery rate of the core. To cover this case, the hypervisor tracks the rate of error recovery for each core. If the rate of error recovery exceeds the threshold, the hypervisor will checkstop the processor. This triggers the alternate processor recovery flow for the system. Even though the processor will have checkstopped when in hypervisor mode, the hypervisor will recognize this as a special case (a hypervisor-initiated checkstop) and will not terminate the system. The normal flows for CPU (central processing unit) sparing, and for policies that are to be deployed when there is no spare, will be followed.

Hypervisor maintenance interrupt handling

Many of the conditions discussed earlier are reported to the hypervisor via the HMER and an HMI. The conditions may be individually masked (enabled) by the HMER. The HMI is blocked when in hypervisor mode, and external interrupts are disabled. Normally, host firmware is responsible for polling the HMER conditions when in this state, but some special cases have a mode-bit enablement to escalate to machine check when the HMI is blocked.

Processor checkstop handling

For processor checkstops, the hypervisor must log out required fault-isolation information and analyze it to determine whether CPU sparing is possible, i.e., whether the checkpoint is intact, or whether the task and partition running on the failed processor must be terminated.

CPU-sparing support

IRR is effective for transient (soft) errors, but not for permanent (hard) errors, since a persistent error will prevent the processor from making any forward progress. Because the RU checkpoint pipeline lags the execution pipeline, hard failures that prevent further execution of instructions usually do not mean that the checkpoint is corrupted. Thus, the checkpoint could be extracted and restored on a different processor to continue executing from the checkpoint. If a spare processor is available in the configuration, then there is no loss of workload or capacity. The checkpoint could also be restored on a previously active processor to recover the workload, although there would be a loss in overall system capacity, which the hypervisor communicates to the OS.

In the event that the checkpoint is damaged because of the error, then the workload cannot be recovered, and the

hypervisor must attempt to contain the loss of the processor to the partition that was running on it at the time of the checkstop.

In order to restore the checkpoint on an alternate processor, it must first be extracted from the defective processor. It is then reestablished by using firmware on the spare processor.

The integrity of the checkpoint is determined by analyzing the error-reporting registers for the failed processor core. If valid, the checkpoint is then loaded into the spare processor, and instruction execution resumes from the checkpoint. This process sequence is performed by the host firmware, similar to a return from a machine check. If the checkpoint integrity is compromised, then sparing is aborted and the hypervisor will attempt to contain the damage to the partition that was running at the time.

Noncheckpointed facilities

Some SPRs are not included in the RU checkpoint and are, instead, extracted by software or otherwise dealt with by the hypervisor in preparation to resume the alternate processor.

Performance monitor

The performance monitor is dramatically distorted after a system-level recovery action, such as CPU sparing, so there would be only very limited value in attempting to restore the data values from the failed processor. Because the overhead to provide logout capability is high, the performance monitor facilities are invalidated instead and are disabled on the spare processor. If performance monitoring was active on the failed processor, a message is posted that it was terminated and should be reinitialized by the user.

Timing facility

The timing facility (TB register, decrementers, and utilization registers) remains active during recovery, so it is not checkpointed in the RU. The TB is synchronized across all processors, so it does not need to be restored on the spare. The decrementer and utilization registers are accessible via a JTAG (IEEE Standard 1149.1) interface and are transferred to the spare processor. The values may have to be adjusted to account for the elapsed time, which can be determined by logging the TB from the failed processor (close to the decrementer and PURR and SPURR values) and comparing it with the TB on the spare processor at the time the timing facility registers are to be restored.

SLB

The hypervisor must invalidate the SLB on the spare processor with an SLB invalidate all instruction (`slbia`) for both threads and restore the SLB contents from the shadow buffer, as is done for an SLB parity error or asynchronous machine check with the SLB-modified bit active in the HMER.

Error-reporting structure and FIRs

Each error detected is fed through a commonly used error-report macro. If sufficient area or wiring is not available to support individual error-report macros for each specific error, then errors are grouped into classes.

The macro is configurable and contains, at a minimum, a mask latch and a hold latch. The mask latch can be used if an error checker is determined to be faulty itself, especially in early hardware debug. A hold latch is used to determine which error (or class of errors) caused a recovery or checkstop event by scanning out of the processor all error-report state latches. In the typical configuration, the hold latch is a set-only latch and does not directly drive the error-output signal of the macro. There are several possible configurations with the macro, but the minimal one is most commonly used.

In more complex arrangements, the hold latch can be inline with the error-output signal if timing requirements dictate such an arrangement. In this case, a keeper latch can be used by one or more of these macro types to provide the memory function usually provided by the hold latch. If more than one error-report macro shares a keeper, the error lines from each are ORed together such that isolation is limited to the class of errors driven by the multiple macros.

This structure is highly effective in debugging early hardware failures, since generally only one error-report macro will be active at a time during hardware validation. In an operational system, however, there will be multiple error-report hold latches active, as recovery may occur multiple times in a healthy machine for several valid error conditions.

The results of the error-report macros are collected together into classes, ORed together, and sent to the FIRs. The classes (e.g., dataflow error, register file correctable error, register file uncorrectable error, and functional) exist for each functional unit in the FIR domain. Each FIR bit has an associated mask bit and action bits that determine what should happen following any occurrence of the associated error type. Allowed actions are recovery (IRR), system checkstop, local checkstop, HMI, or machine check. This allows errors in a hardware validation environment and in a production environment to be treated differently, or for a faulty error class to be completely masked off. In the case of checkstops, further FIR bits are prevented from being set, since cascading errors may occur, and their recording would confuse the isolation of the checkstop-causing error.

The processor core contains two FIRs, one for instruction-side events and one for execution-side events. The nest logic contains multiple FIRs, one for each logical unit (e.g., the L2 quadrant, L3 directory slice, external bus controller, fabric bus interfaces, and memory controller). The FIR bits are cleared in the process of IRR.

Because a single error may cause multiple cascading errors, two *who's-on-first* (WOF) registers are provided to capture the source of the first error detected in the processor. Once any error occurs and is recorded in the WOF, all subsequent errors are blocked from setting bits in the WOF. Which WOF is selected toggles when a FIR reset occurs, making available the two most recent sources of IRR.

Concluding remarks

The POWER6 processor design includes robust error checking and facilities for IRR that cover most errors that can occur. This functionality is accomplished through checkpointing the execution flow, along with other mechanisms used for special cases and the L1, L2, and memory storage structures. The POWER6 processor design also provides the capability to transfer the working state of a processor core to another spare core in the system, when the original core is deemed to have a persistent error condition. A reduced mode of IRR, minirefresh, can be used to increase the performance of certain types of operation and is also used in early hardware debug workarounds.

Using virtual storage keys to improve OS reliability is just one application of the virtual page class key protection function. There are many more, such as sharing privileged data with an application. Even a malicious application would not be able to break the protection, since it does not have access to the AMR. The hypervisor may use the facility to enable controlled sharing of data among multiple partitions (especially a partition and a program providing services to the partition). More applications of virtual storage keys are planned to be developed over time.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of SPARC International, Inc., in the United States, other countries, or both.

References

1. Power.org, Power ISA™ Version 2.04; see *http://www.power.org/resources/downloads/PowerISA_203.Public.pdf*.
2. J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, et al., "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)," *IBM J. Res. & Dev.* **40**, No. 1, 3–18 (1996).
3. H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, et al., "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor," *IEEE J. Solid-State Circuits* **38**, No. 11, 1896–1905 (2003).
4. M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi, "RAS Strategy for IBM S/390 G5 and G6," *IBM J. Res. & Dev.* **43**, No. 5/6, 875–888 (1999).
5. D. C. Bossen, A. Kitamorn, K. F. Reick, and M. S. Floyd, "Fault-Tolerant Design of the IBM pSeries 690 System Using POWER4 Processor Technology," *IBM J. Res. & Dev.* **46**, No. 1, 77–86 (2002).

Received January 17, 2007; accepted for publication May 30, 2007; Internet publication October 23, 2007

Michael J. Mack *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (mjmack@us.ibm.com).*
Mr. Mack is a Senior Engineering Manager in processor development. He received a B.S. degree in computer engineering, *summa cum laude*, from Syracuse University. He joined IBM at the MIT Laboratory for Computer Science, developing hardware for parallel-processing research. He has worked on processor designs for System/370*, AS/400* and RS/6000* machines as well as on various system-on-chip designs for external customers and IBM standard products. Mr. Mack joined IBM Austin to work on the POWER6 processor project as the recovery unit lead and later as the core pervasives unit lead.

Wolfram M. Sauer *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (wsauer@us.ibm.com).*
Mr. Sauer is a Senior Technical Staff Member in the processor development area. He received a diploma degree ("Diplom-Informatiker") in computer science from the University of Dortmund, Germany, in 1984. He subsequently joined IBM at the development laboratory in Boeblingen, Germany, and worked on the S/370* (later S/390* and zSeries) processor design, microcode, and tools. He joined IBM Austin in 2002 to work on the POWER6 processor project.

Scott B. Swaney *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (sswaney@us.ibm.com).* Mr. Swaney received a B.S. degree in electrical engineering from Pennsylvania State University. He joined IBM in the Enterprise Systems Division as a VLSI (very-large-scale integration) design engineer. He is currently a Senior Technical Staff Member working on hardware and system design for the IBM eServer*. Mr. Swaney specializes in design for high availability in microprocessors and holds multiple patents related to processor recovery.

Bruce G. Mealey *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (mealey@us.ibm.com).*
Mr. Mealey is a Senior Technical Staff Member in the OS development area. He received a B.S. degree in electrical engineering from the University of Texas. He joined the IBM Development Laboratory and has worked on system performance, system bring-up, and OS development. Mr. Mealey is currently working on OS reliability, serviceability, and availability.