Enhanced I/O subsystem recovery and availability on the IBM System z9

K. J. Oakes
U. Helmich
A. Kohler
A. W. Piechowski
M. Taubert
J. S. Trotter
J. von Buttlar
R. M. Whalen, Jr.

Although part of the IBM System z™ strategy is to improve design and development processes to prevent errors from escaping to the field, improving recovery is another element in the strategy to keep a machine up and running should an error occur. The z9™ continues on an evolutionary path of enhancing I/O subsystem (IOSS) recovery to further advance the reliability, availability, and serviceability (RAS) of System z platforms. This paper presents an overview of recovery and how it interacts with other RAS functions—such as error-detection mechanisms in hardware, including automatic identification and recovery of failing elements—up to the point in time prior to the advent of the z9. It then presents the innovations to IOSS recovery and error detection in the z9 that further improve machine availability. The recovery infrastructure, which significantly reduces recovery time and makes recovery much less dependent on machine scaling for this and future generations of System z servers, is described. Also described are such innovative uses of this new infrastructure as improvements in error detection related to elusive firmware problems seen in prior machines, the ability to detect and recover from firmware hangs or lockups related to inadvertently leaving control blocks locked, and the capability to perform recovery in parallel by multiple system-assist processors.

Introduction

Over the years, System z* (formerly zSeries*) servers have been continuing to scale [1, 2] in terms of the sheer number, capacity, and performance of their logical partitions, processors, channels, networking adapters, and the I/O devices to which they can attach. As a result, customers have been placing a greater percentage of their business-critical workloads on these machines. The expectation of uninterrupted, around-the-clock operation has made it increasingly important to minimize failures that affect the operation of the machine. The z9* continues in the System z evolutionary path to the highest quality standards in both the hardware and firmware [3–5]. However, statistics such as mean time to failure or failure rate, no matter how good, still allow for errors. And just one failure that takes a machine down could have a severe impact on a client's business. The engineers at IBM recognized this and, in mainframes as early as the IBM 3081, announced in 1980, they started on a continuous path of delivering innovative solutions to improve the reliability, availability, and serviceability (RAS) of the machine to unprecedented levels in the industry. One such innovation, serving all three aspects of RAS, was the concept of *recovery*—the ability of a machine to automatically restore itself to a known state after taking an error and, if need be, to fence off failing components so that the machine can continue running on the unfenced portions of the machine with minimal disruption to the running software.

In a System z eServer*, recovery code and hardware are spread over the entire system [4, 5], ranging from primary components, such as the first-order I/O network, I/O

0018-8646/07/$5.00 © 2007 IBM

channels, or processor chips, to secondary components, such as the power subsystem or the support element (SE). For the functional components, recovery is implemented mainly in firmware, which runs on the central processors (CPs) and the system-assist processors (SAPs). Especially in the I/O subsystem (IOSS), a significant portion of the functionality is implemented in firmware [6]. As a result, recovery must deal not only with hardware failures, but also with firmware errors and a huge number of data structures that reflect the current state of operation.

The typical recovery goals [3] and sequence of steps are as follows:

1. *Error detection:* Detect the error as early as possible through the use of appropriate checking mechanisms. This applies to hardware failures and firmware errors.
2. *First failure data capture (FFDC):* Collect comprehensive information on the error itself and on the circumstances that led to the failure. This step lays the foundation for all subsequent recovery, service, and repair actions dealing with the error, as well as for subsequent analysis by System z development groups.
3. *Determining and scheduling of recovery actions:* After analyzing the FFDC data, determine the recovery actions that must be performed in which order for which elements in the system.
4. *Execution of recovery actions:*
   - *Reset of affected elements:* Reset the elements affected by the error to bring them back to a known and consistent state that allows resumption of normal operation. This applies to hardware elements and the firmware stack. A hardware element can be restarted successfully only if it is not permanently damaged, i.e., if the error can really be "repaired."
   - *Fencing of failing elements:* Fence off the failing element in case it is permanently damaged. This is to protect the system as a whole from a continuously malfunctioning element.
   - *IOSS control block (CB) recovery:* Examine the hardware system area (HSA) for I/O CBs that were in use by affected elements at the time of error and restore them to a known, consistent state, thus making them available for continued usage.
   - *Retry/termination of affected operations:* Either retry or terminate all operations upon which the failing element was working when the error occurred. If a retry is possible, the recovery action is transparent to the software. However, depending on the error, there may be no way to tell how far the failing element progressed in executing the operation. In this case, the operation must be terminated because, for example, firmware internal data structures may have become inconsistent, and they must be reset to a known state.
5. *Software notification:* Notify the customer's software that an error has occurred and which operations have been terminated. Again, depending on the error, there may be no way to tell how far the failing element progressed in executing the operation. Therefore, software usually has to perform its own recovery actions in order to allow the resumption of normal operations. For example, when a failing operation is part of a transaction, this may include rolling the transaction back.
6. *Serviceability infrastructure notification:* Notify the System z eServer SE that an error has occurred. This includes submitting and logging all available FFDC data and the state of all affected elements after completion of the recovery actions. On the basis of this information, the SE determines which component of the system must be repaired and triggers the service personnel to schedule the appropriate repair action.

The following paragraphs briefly describe some of the basic concepts of error detection, recovery, and analysis.

Error detection

Comprehensive error detection in hardware and firmware is indispensable to the performance of a successful recovery [3]. Detecting errors as early as possible is vital to the avoidance of data integrity problems. In a System z server, error-detection capability, like recovery functions, is present among all system components. In hardware, a wide variety of mechanisms are implemented as appropriate to the specific chip function. Some examples are parity checks of registers, cyclic redundancy checks (CRCs) for serial data transfers, error-correction code (ECC) logic in combination with thresholding, validity checking of state transitions for state machines, consistency checks at interfaces, and comparison of results of double execution units.

Available literature (e.g., [7]) discusses these mechanisms. For detecting firmware errors, far fewer options are available, because there is no way to really "test" the code itself at runtime. Here, error detection relies solely on active checking for bad parameters or inconsistencies in the data used by the code.

Error data collection

On System z mainframes, an important aspect of the recovery infrastructure is error data collection. FFDC

mechanisms to collect unpolluted data on the original failure are required to classify an error and to determine the potential source of the error. It is critical to collect this data as close in time to the detection of the error as possible. While handling the error (e.g., sending controls to failing hardware to reset error indications), more data is collected to record the effect of each action performed during the recovery task. This mechanism is known as *secondary failure data capture* (SFDC).

FFDC and SFDC are tied closely into the general recovery infrastructure, but may also be used by modules other than recovery. While recovery code uses this data internally to develop a recovery strategy to handle the current problem, the data is also logged into permanent storage on the SE hard disk. This data is used by automated mechanisms on the SE—e.g., to identify a failing field-replaceable unit (FRU)—and sent to IBM System z support. In rare situations, the automated error-handling mechanism cannot resolve the failure situation as expected. The data collected in these situations is then made subject to a detailed postmortem analysis by the development team.

Error classification

Each error has specific characteristics that imply whether and how recovery for it can be performed. There are numerous meaningful criteria for classifying errors. Many of them are reflected in the algorithms implemented in a System z server [3] that determine the appropriate recovery action for a given error. The most important ones are discussed in the following subsections.

Clock-running and clock-stop errors

The most important goal of a System z server is to preserve the integrity of the customer's data. In severe error situations, satisfying that goal may mean that it is required to immediately stop a piece of hardware, or even the whole system. Such errors, called *clock-stop errors*, cannot be recovered. Fortunately, the mean time to failure (MTTF) for System z servers is very high (30+ years), so the occurrence of such an error is a rare event, and only a small number of errors that occur require such a drastic step. In "normal" error situations, the integrity of the customer's data is not endangered, and recovery takes all required actions to keep the system operational. These errors are therefore called *clock-running errors*.

Hardware failures and firmware errors

A piece of hardware may age and fail over time. In this case, the System z recovery first attempt to repair it involves resetting and reinitializing the failing hardware. If this is not successful, the broken piece of hardware is isolated to protect the remaining system from potential misbehavior of the damaged element. This is done by fencing it off, i.e., by shutting down all interfaces to the failing element.

Either firmware errors exist in the code or they do not. Firmware does not age, so it cannot break over time, and errors usually surface as soon as the faulty code is executed with a combination of conditions (such as input parameters, stimuli, or workloads) that precipitate the fault. If a System z eServer detects a firmware error, the currently active task is usually aborted by restarting the complete firmware stack on the processor that executed the faulty code. Whether or not a successful recovery is possible depends on the damage that has already been caused to the firmware data structures. If they can be reset to a consistent state, normal operation is resumed; if they cannot, execution of all system functions that use the damaged data structures must be inhibited. In severe cases, this can result in the entire system being stopped.

Of course, the recovery code itself is a piece of firmware and may therefore contain errors. If such an error surfaces during execution, it means that the recovery algorithms are unable to deal with the error situation that initiated the recovery. There is no more-sophisticated code to handle this situation; if there were, it would already have been part of the initial recovery code. Therefore, in such situations, the entire system is stopped. The System z development group puts a very strong focus on robustness and quality of the recovery code because this firmware component, by its very nature, executes in situations in which some error within the system has already caused a certain amount of damage. Therefore, the recovery code never relies on data structure content that is also used by functional code. It also contains many more consistency checks than functional code.

Recoverable and permanent errors

Most errors surface only in very specific circumstances. Usually it is possible to reset the failing element, repair the damage caused by the error, and resume normal operation. Such errors are called *recoverable errors*. However, if a piece of hardware has a solid, non-intermittent failure due to aging or if it is not possible to restore a consistent system state after an error, the failing element must be fenced off. Such errors are called *permanent errors*. Of course, any error that has been recovered successfully might reoccur. For example, a piece of hardware may show the same recoverable error multiple times before it finally breaks permanently. It is likely that a firmware error will show up repeatedly, since the specific circumstances precipitating it typically reoccur. If an error persistently and rapidly reoccurs, the resulting recovery actions could monopolize vital system resources, such as processors or I/O paths. This could have a severe impact on the availability or performance of the system. To avoid such situations, all recovery actions

and errors are counted for each element in the system. If an element fails repeatedly with the same recoverable error so that the appropriate counter exceeds a predefined threshold value, the error is treated as if it were permanent. As escalation, the element is fenced off to protect the remainder of the system. In the case of a firmware error, *fencing* means inhibiting further execution of the affected function by maintaining disablement flags and having appropriate checks to the code. Obviously, this is applicable only to nonvital functions.

Initial and secondary errors

In complex computer systems such as a System z server, an initial error that occurs in one element often causes additional follow-on errors to occur in other elements. This effect is known as *sympathy sickness*. The required recovery actions for an error are independent of whether it is an initial or a secondary error. However, after recovery is complete for a hardware failure, this difference is highly important for determining which hardware component must be replaced. Using the FFDC data collected during a recovery run, the FRU error analysis first attaches predefined weights to all available error indications. The element with the highest weight is the one that must be replaced. The algorithms used in this process are based on the relevance of the individual error indications, the hierarchy of the way in which chips are connected, and the information concerning the success of the recovery or the lack of it.

Scheduling of recovery actions

After an error is detected and FFDC data is collected, recovery must determine the elements in the system that are affected, the recovery actions that must be performed, and the order in which they must be performed. For errors affecting only one element, this is usually straightforward because there are no dependencies. If an error affects multiple elements in the system, the scheduling becomes much more complicated. This often occurs if an initial error in one element causes other elements to detect follow-on errors, e.g., because of timeouts or because some required resource is no longer accessible. In the IOSS, many errors are of this kind because of the chip hierarchy in the first-order I/O network. Elements closer to processor chips must be recovered successfully before elements farther away can even be scanned to determine whether they have encountered an error. This requires that the execution of the recovery actions for the first-order I/O network components be, in fact, interlaced with the FFDC step.

The scheduling step also determines which processors should perform which recovery actions. In a System z server, SAPs perform the IOSS recovery. There is a static assignment of which SAP is responsible for performing the recovery for a given I/O channel. Multiple recovery requests that are independent of one another are distributed over the SAPs that are affected by the errors and are executed in parallel.

It is important to determine when to initiate the IOSS recovery. Most resources and data structures are shared among multiple elements such as channels or processors. If a resource or data structure is held or locked by an element, the recovery code is allowed to "steal" it only if there is a recovery request for the owning element. Because of this, starting recovery too early may result in deadlocks if the complete picture of all pending errors in the system and the required recovery actions is not yet known. However, starting too late, when resources may be unavailable, may cause additional follow-on errors, thus endangering the success of the overall recovery. The parallel recovery execution section below describes the scheduling in more detail.

Execution of recovery requests

The execution of a recovery action concentrates on either bringing a failing element back into a consistent state in which normal operation can be resumed, or isolating it from the rest of the system.

This action consists of the following steps:

1. *Reset and reinitialization of failing elements:* Most errors that can occur in a System z eServer are considered to be recoverable in the first attempt. For hardware failures and firmware errors such as programming exceptions and deadlocks, the recovery code attempts to reset and reinitialize the failing element. The firmware stack running on the affected processor is restarted. For most hardware elements, multiple levels of resets are available, with their scope ranging from only a small section of chip logic to the complete chip. An initial recovery attempt always uses the least invasive reset type. If it is not successful (i.e., the subsequent reinitialization of the element fails), this indicates either that the hardware is permanently damaged or that the scope of the reset was not broad enough to cover all affected hardware pieces. In both cases, recovery begins a retry, including error-data collection. The recovery actions performed in the retry are more comprehensive than those in the initial attempt:
   a. If new errors show up, they were probably the cause of the failure of the first recovery. The recovery actions derived from them are added to the initial ones.
   b. If there are no new errors, the retry uses a more comprehensive reset that covers a larger portion of the hardware, up to a full chip reset. If the previous

recovery attempt already involved a full chip reset, the error cannot be recovered. In this case, the failing element is fenced permanently.

2. *Fencing of failing elements:* Fencing of a failing element is done if the reset and reinitialization does not work or if it encounters one of the few error types that by their nature cannot be recovered. For that, recovery isolates the failing element, for example, an I/O channel card, from the remainder of the system. The goal is to avoid a continuously malfunctioning element that permanently disturbs the operation of the system, since this would, at the least, have a performance impact on the customer's system. In extreme cases, it could result in the machine being completely occupied by recovery and no longer available.
3. *IOSS CB recovery:* The HSA is a memory space that contains state information for all I/O operations occurring in the processor complex. Thus, maintaining the integrity of this memory space is essential to the operation of the entire system. The HSA is organized into sections, with each section containing a particular type of IOSS CB. Each CB type is a logical representation of particular physical resources of the IOSS configuration. These individual CBs are used by the IOSS firmware to manage I/O operations; they contain state information related to the I/O operations executing in the IOSS. If the firmware has to update a CB as part of processing an I/O operation, the element, such as a SAP or a CPU, acquires exclusive use of the CB by using a locking protocol. It acquires exclusive access to the CB until the point at which it releases or unlocks the CB. During the time interval between the time at which the element acquires the lock and the time at which it releases it, the CB is considered to be held by the element and is in a transitional state. If the element holding the CB lock fails, the CB requires recovery to restore it to a known state and make it available for use by the other elements. Each IOSS CB type has a unique recovery algorithm designed to recover the associated CB type during IOSS recovery actions.
4. *Retry or terminate ongoing operations:* Reset or fencing of a failing element affects all of the operations that this element had been working on when the error occurred. However, the IOSS internal data structures still reflect these operations and have to be reset. Depending on the type of operations, it may be possible to retry them. If so, the recovery is transparent to the customer's workload. Since other operations cannot be retried, they must be terminated; in these cases, software must be notified.

Software and service infrastructure notification

For all operations that, because of a recovery action, must be terminated according to the z/Architecture*, software is notified. On the basis of these notifications, the operating system (OS) may also have to perform some software recovery action. When subchannel operations are terminated, z/Architecture [8] provides for the subchannel to be put into a channel-control-checked (CCC) state, which is seen by the OS when the subchannel is made *status pending*. When channels are put into a *permanent error* or *check-stop state*, channel report words (CRWs) are presented to the OS by the machine-check interruption mechanism [8].

Furthermore, the service infrastructure is informed about the recovery. The IOSS firmware sends all collected FFDC data and results of the recovery actions to the SE. On the basis of this data, the SE performs several tasks:

- *Customer notification:* The SE sends the customer a hardware message concerning the error. There are also OS error messages, but customers do not necessarily monitor all consoles.
- *FRU analysis:* The FRU analysis consists of analyzing the FFDC data for clock-running errors to identify the failing element that caused them.
- *Overall problem analysis:* The overall problem analysis (PA) combines the results of the clock-running FRU analysis with all other error indications in the system, such as clock-stop errors, SE errors, or power subsystem errors. On the basis of this system-wide overview of all failures that occurred within a certain time window, the problem analysis routine identifies the FRU that was the root cause of the errors.
- *IBM service personnel notification:* The FRU identified by the PA is automatically reported to IBM by what we term a *call home*. If a service contract is in place, this triggers IBM service personnel to contact the customer and schedule a repair action. In a System z eServer, a repair for a clock-running error can usually be performed concurrently, i.e., without further disruption of the customer's workload.

IOSS CB recovery overview

Before the advent of the z9, the IOSS CB recovery design used a generic scan method executed on a SAP to perform recovery. As the name implies, the scan method examines the entire HSA where all of the IOSS CBs reside. It scans one CB at a time searching for the few CBs that are held by the failing processor unit (PU). With large I/O configurations, tens of thousands of CBs are examined during this process, which can greatly extend the time required to recover from an error.

Long system recovery times have a direct impact on overall system performance. Recovery tasks have a higher internal priority, which delays other tasks that may require the SAP performing the recovery action. Other tasks executing on running PUs may require one or more of the CBs scheduled for recovery. These tasks have to wait for the held CB to be recovered and freed, which can drive additional recovery actions (i.e., timeouts and CB hang conditions). The large number of PUs in the system configuration contending for a given set of shared CBs only exacerbates this limitation.

The z9 enhanced IOSS recovery design introduces the concept of CB state tracking. This method has significantly reduced system recovery times and improved overall system throughput and performance; it has resulted in predictable, near-constant recovery times instead of scaling with the size of the I/O configuration.

IOSS CB introduction

During the initial machine load (IML) of the z9 central processor complex (CPC), firmware reads the installation-generated I/O configuration dataset (IOCDS) and translates it into I/O CBs, such as the subchannel control block (SCB), in HSA. Many types of I/O CBs are required to logically define the IOSS, with each type defining a particular component of the IOSS. Thus, the HSA is memory space containing state information for all of the I/O operations occurring in the CPC.

IOSS tasks and modes of operation

From an OS viewpoint, a task may be seen as a piece of work or job stream managed by the system-level software. The system firmware also views work on a task basis, but the definition is firmware-code-specific. The z9 CPC comprises up to 64 PUs, and each PU can be assigned one of two functional modes of operation:

1. As a CPU, executing software in z/Architecture or Enterprise Systems Architecture/390* (ESA/390) mode [8], with the capability of entering millicode [9] mode to execute specific instructions. In this mode, the scope of an IOSS task is defined as the execution of that instruction by millicode.
2. As a SAP, executing firmware, primarily in i390 [6] mode. A SAP views a task as a dispatchable unit of work—for example, selecting a channel path to be used for an I/O operation and initiating the operation at that channel.

Within a task on a SAP, i390 code also invokes millicode in certain situations, such as broadcasting an interrupt condition to the CPUs. The firmware definition of a task and its scope are the basis for understanding state tracking and the IOSS CB recovery design of the z9 IOSS.

CB locking protocol and state tracking

Most CBs are accessed by different elements in the system, namely CPUs, SAPs, and channel paths. Lock fields in the CBs are used to implement an access protocol. There are also a number of locking rules, such as a strict order in which locks are to be acquired. The order ensures that no deadlocks occur. If, for instance, SAP A locks CB 1, then SAP B locks CB 2, and then SAP A attempts to lock CB 2 and SAP B attempts to lock CB 1, both SAPs are deadlocked. Therefore, CBs are assigned priorities according to their type.

Another rule states that all locks may be held for only the duration of a single task, such as dispatching a new I/O operation or passing the status of such an operation back to software upon completion. If a lock is inadvertently held for a longer time, it may go unnoticed until that CB is required again in a subsequent task.

If a task in the IOSS is aborted because of a hardware or firmware failure, it is likely that locks are still held. To recover from this situation, IOSS code on previous systems would scan all CBs to find out which ones participated in the aborted task. As the number of CBs grew into the tens of thousands, this process took on the order of one second to complete. On the other hand, a single task will access only a very limited number of CBs, usually less than ten. For this reason, state tracking was introduced in the IOSS of the System z9*.

All lock operations are now tracked in a separate task control block (TCB). The CPU maintains its own TCB, as does the SAP; moreover, i390 code and millicode maintain separate TCBs. This way, there is no requirement to lock the TCB itself, which would have created unacceptable contention. The TCB contains a number of slots that correspond to the maximum number of CBs being locked at any given time. When a CB is locked, a free slot is picked in the TCB in which the CB address and type are recorded. The slot number is stored in the CB itself. This avoids an expensive scan of the TCB when the CB is unlocked. To safeguard against programming errors, the locking routine validates that the CB is not already held by the processor requesting the lock, which would be a violation of the locking rules.

CB locking and the TCB

Each PU is allocated two TCBs, one for i390 mode and one for millicode mode. The TCB is a registry of the CB resources currently owned by a task executing on the processor.

When a SAP operating in i390 mode is dispatched to perform a specific task requiring CB resources, it invokes a locking protocol to acquire the proper serialization for

updates to that CB. **Figure 1(a)** depicts an IOSS CB with its owner lock data and extended lock information fields used by the locking protocol to identify the type of lock, the processor, and information about the task holding the lock.

The locking protocol also provides the common i390 code interface to the TCB infrastructure to record important information about the CB. **Figure 1(b)** depicts a TCB. The TCB contains several arrays to maintain information about the CBs locked during a task. Its key fields are the following:

- *CB mask (CBM) array:* This array relates to a corresponding entry in the CB type (CBT) and CB address arrays (CBA) which identifies a locked CB or one intended to be locked by a task.
- *CBT array:* This array contains the CBTs of the CBs that are locked or intended to be locked by a task. Each CBT entry corresponds to a bit in the CBM array.
- *CB address array:* The CBA array contains the storage address of the CBs locked or intended to be locked by a task. Each CBA entry corresponds to a bit in the CBM array.

The locking protocol updates the TCB arrays with the CB information and also updates the CB with the TCB CBM array index. The index is used by the unlocking protocol code to quickly locate the correct TCB array entries associated with the CB.

After the CB state information is updated, the CB can be released using the unlock protocol. The unlock protocol resets the specific CBM array bit to indicate that the entry for this CB is no longer valid.

Enhanced error detection by locking protocols

The locking and unlocking protocols make use of the TCB infrastructure to detect error conditions with respect to control usage by a task or errors that exist in the HSA. The firmware code contains a set of rules that apply to and support the tasks based on the structure of the IOSS. The rules also are used to verify that locking protocols relative to task execution are observed by the firmware.

Task scope checking

All locks may be held only for the duration of a single task. At the start of a new task, the IOSS interrupt handler interrogates the TCB CBM bits. If there are any bits set, this indicates that the previous task did not fully unlock the CB resources it held for that task. In this case, a microcode-detected error is set, I/O recovery is scheduled, and firmware debug data is collected and submitted to the SE for analysis.

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	Owner lock data		Extended lock information	
1	CB type			
.	CB specific state data			
.				
.				
.				
.				
n				

(a)

Word	Byte 0	Byte 1	Byte 2	Byte 3
0				
1	TCB type	PU#	0	
2	CB mask (0:15)			
3	:			
4	Task footprint			
5–6	:			
7–11	Extended error information			
:	:			
28	CB type(00)	CB type(01)	CB type(02)	CB type(3)
29–30	:			
31	CB type(12)	CB type(13)	CB type(14)	CB type(15)
32–33	Control block address (00)			
:	:			
62–63	Control block address (15)			

(b)

Figure 1

Control block definition: (a) common IOSS CB; (b) TCB.

Double unlock of a CB

The unlock protocol stores back the CB data, then updates the CB lock word, thus freeing the CB for use by another processor. If a task attempts to unlock a previously unlocked CB, the unlock protocol detects this condition by checking the CBM bits, thus preventing a condition in which the CB could have been overwritten, damaging the HSA data space.

CB verification

The unlocking protocol verifies that the CB being unlocked is the correct one. The TCB CBT array entry is compared to the CBT in the CB, and the TCB CBA array entry is compared to the CBA. If there is a mismatch, a microcode-detected error is set, I/O recovery is scheduled, and firmware debug data is logged for analysis.

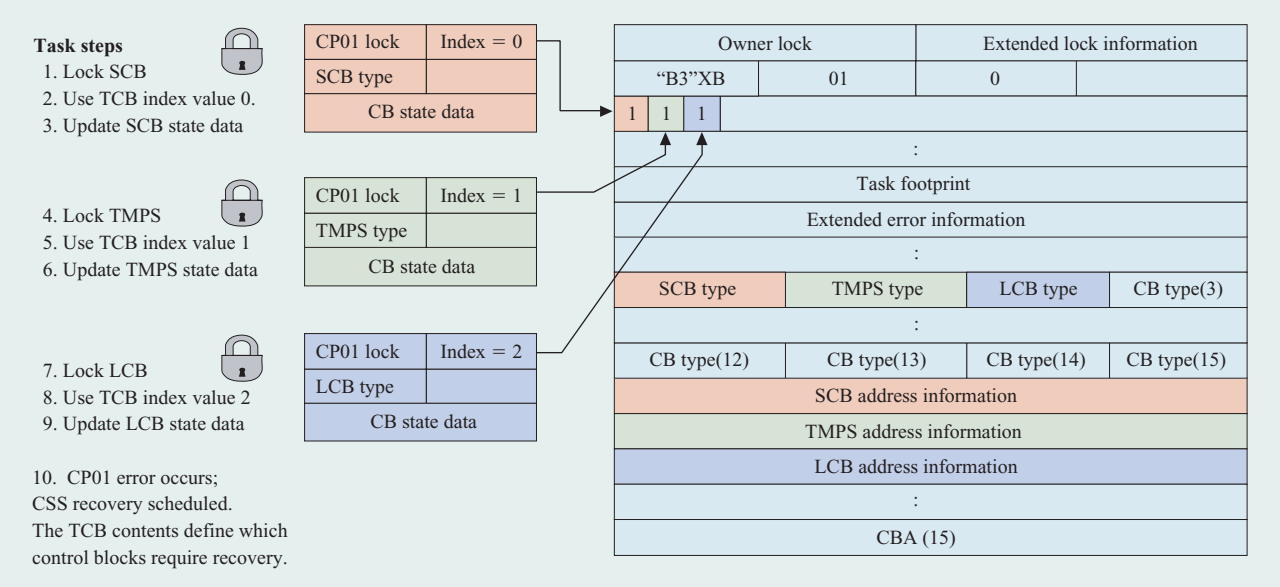

Figure 2

Task steps populating the TCB; then an error occurs.

Early hang detection

When a SAP or a CPU firmware detects an invalid or unexpected condition, it requests recovery by a SAP. This request may be made pending while the SAP is in the middle of a task and thus (yet) unaware of this request. If the SAP subsequently attempts to lock a CB that is currently held by the failing SAP or CPU, a deadlock occurs. This situation was formerly detected and resolved by a timeout mechanism.

A different approach is used in the System z9. When SAP A detects that it is unable to lock a CB because it is being held by SAP B, it checks whether SAP B has requested a recovery from SAP A. In this case, SAP A immediately invokes recovery without any delay.

Finally, when an IOSS task completes, no CBs are left locked. At the start of the next task, the slot usage vector in the TCB is checked. If it is nonzero, a lock has not been released by the prior task. This triggers a recovery action, and firmware debug data is collected and submitted to the SE for analysis.

IOSS CB recovery use of the TCB

In the event of a PU failure related to either hardware or firmware, the TCB has information about the CBs in use by the PU at the time of the failure. These CBs require recovery because they are the CBs whose state information was being altered when the failure occurred and may now be in an indeterminate state. IOSS CB recovery consists of a set of algorithms, with each algorithm unique for a particular CB type. The algorithms are designed to interpret the TCB contents, restore the targeted IOSS CBs to known states, and make them available for future tasks.

Figure 2 depicts a task executing on CP01 that has locked three CBs during its execution before taking an error. IOSS recovery will be scheduled, and the TCB content used to direct the recovery processing.

Early in the IOSS recovery sequence, the TCB is validated to ensure that its contents are accurate. The CBM array is checked to see whether the entries are valid, the corresponding CBT entries are examined for valid CBTs, and the CBA entries are examined for valid CBAs. Address range checks performed on the CBA information ensure that the CB resides within the expected address range for that CB type. Invalid CB entries are considered firmware errors. They cause firmware debug data to be collected, and are removed from the TCB and further recovery processing.

Once TCB validation is complete, the TCB is interrogated by the CB-specific recovery algorithm, which searches for a valid TCB CBM array entry for the specific CB type to be recovered by the algorithm. If a valid TCB CBM entry is found for the CB type, the CB address is retrieved from the TCB CBA array and returned to the recovery algorithm. Once a valid CB address is returned to the recovery algorithm, it can locate the CB and restore it to a known state. This process is repeated for each algorithm until the TCB has been fully processed. Since it is possible for multiple errors to occur simultaneously in

the IOSS, it is possible to have multiple TCBs to process during a single execution of IOSS recovery code.

CB hang recovery

Even though the advanced error-detection methods described above prevent a number of CB hangs from occurring, there is still the need to recover from a CB hang in the event that a firmware error escapes detection. While the new TCB infrastructure was being formulated, it became evident that it could also be an integral part of the solution for recovering from CBs inadvertently left locked by a PU.

Elusive CB hang and its consequences

IOSS CB recovery prior to the z9 works very well for recovering CBs left locked by a PU that subsequently underwent a hardware failure before being able to unlock them. This is because the identity of the locking PU is set into the owner portion of the CB lock word when the CB is locked. This allows IOSS CB recovery to know which CBs were left locked by the failing PUs so it can recover and unlock them. But what happens if a CB was locked by a PU and a firmware bug prevents the PU from unlocking it, but the PU continues to run? In this case, the PU that left the CB locked is typically healthy from a hardware viewpoint; there would be no indications of processor errors. However, the unsuspecting PU that attempts to lock that CB will spin as it waits for that CB to be unlocked by the prior owner until, eventually, a timeout occurs for this unsuspecting PU.

Most tasks within the IOSS are timed, so that if timeout occurs, the PU running the task is considered to be hung. When the PU is running in i390 mode, running tasks are timed by a mechanism called the *watchdog* timer that is external to the task itself; i.e., it does not keep track of the specifics of the task. Hence, when an i390-mode watchdog timeout occurs, it is not obvious whether trying to lock a specific CB caused the hang. The CB or CBs left locked by the PU that "forgot" to unlock it would not be recovered by the current IOSS CB recovery method, as illustrated in **Figure 3(a).** CBs that are inadvertently left locked and cause other elements to hang are classified as *hung CBs*.

Other PUs could also eventually time out attempting to lock this CB, perhaps multiple times, causing multiple invocations of IOSS CB recovery. If a PU is taken through a recovery process multiple times within a certain period of time, there is a recovery escalation of that PU to a check-stopped state, which essentially fences off the PU, making it unusable. If multiple PUs were check-stopped, the system could reach the point of being unusable, or worse, no PUs would be left in an operational state and the entire system would be put into a system check-stop state. Analysis of FFDC from system check-stops that have occurred in the field in past machines as far back as the z990 has revealed cases in which multiprocessor hang scenarios were the root cause.

Recovery from CB hangs in a multiprocessor environment

Hang recovery makes use of the TCB described above and introduces new constructs in a novel way to identify and recover CBs that were inadvertently left locked, causing the hang condition described previously. The basic concept is this: Given that there are TCBs containing addresses of CBs that the PU either locked or intended to lock, hang recovery can examine the locks in each of the CBs listed in the TCBs of the hung PUs to determine which PUs actually had those CBs locked. If any of those CBs were locked by another PU, hang recovery could then examine the TCBs of the PUs holding the locks. Depending on whether or not the PUs holding the CB locks were running or being recovered, hang recovery would then determine what recovery actions to take.

The notion of *intending to lock* is captured in the TCB as a result of the locking rules in managing the TCB and CB. In short, before a PU attempts to lock a CB, it must first set the appropriate CB information in the TCB. Thus, should the PU hang on trying to obtain a lock, the TCB would have the information about the CB that could have caused the hang. In the case of the PU running in millicode mode, the millicode knows exactly the CB on which it hung and explicitly indicates that CB in the TCB after a PU has hung. In either case, the methods of hang recovery described below can be applied to a hang detected either in i390 mode or in millicode mode.

Hang recovery algorithm

Hang recovery uses the TCBs of the PUs it is recovering to examine the locks in CBs indicated in the respective TCBs. If the hung PU held the lock of a CB, that CB could not have been the cause of the hang, and it would be recovered, as is done today by IOSS CB recovery. However, if the lock of another PU was found in the CB and was pointed to by a valid CBA in the TCB of the hung PU, hang recovery would have to perform additional analysis to see whether the fact that that CB was left locked caused the hang.

The next step is to determine whether the PU holding the lock has been scheduled for or is already in recovery. If either is the case, hang recovery examines the TCBs of the PU holding the lock to see whether that CBA was in a TCB of the PU holding the lock. If the CBA is found in the TCB of the PU holding the lock, hang recovery removes this TCB entry from the TCB of the hung PU because it is assured that the locked CB will be recovered

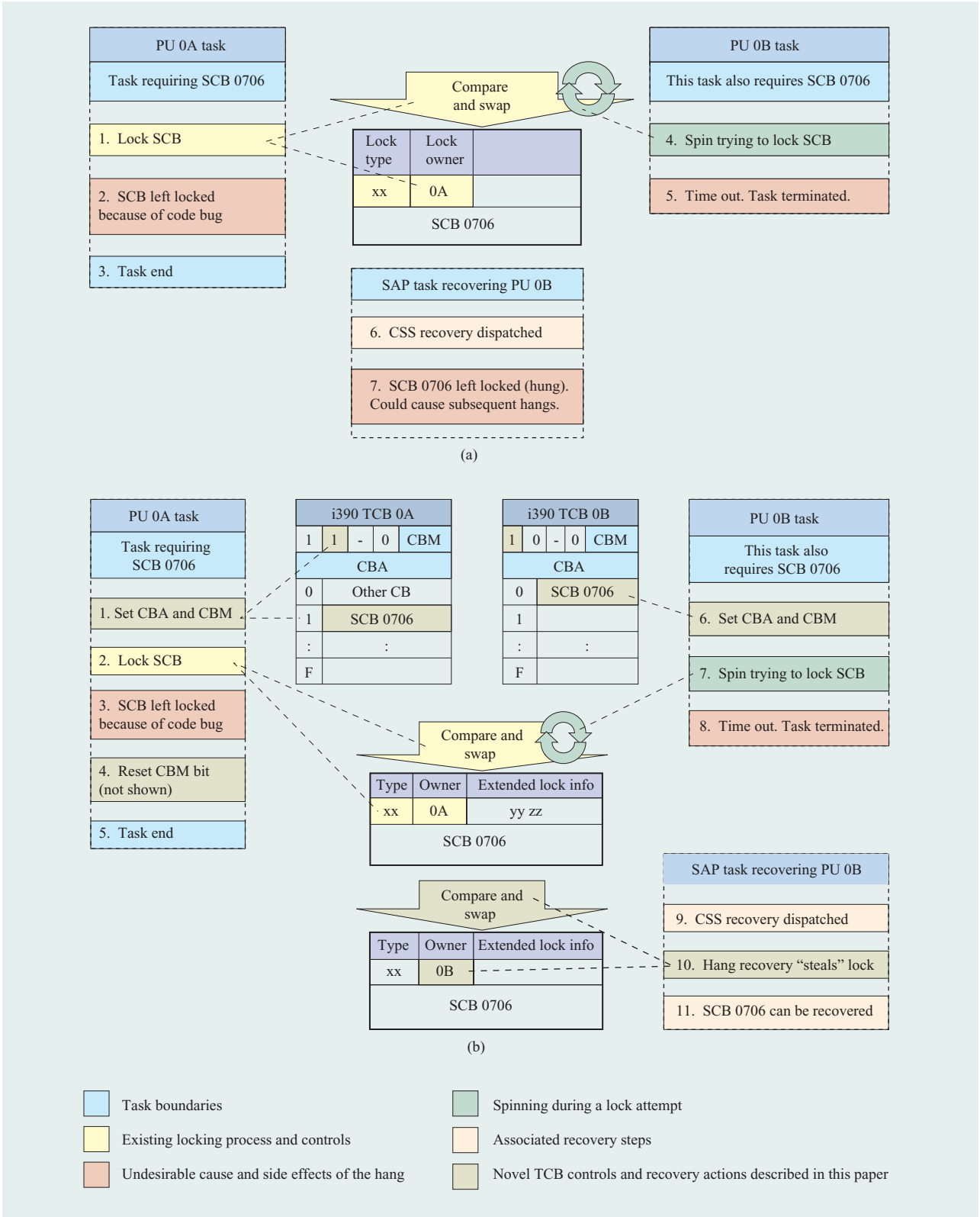

Figure 3

(a) Hung CB; (b) CB hang handled by recovery.

by the task responsible for running IOSS CB recovery against the TCB of the PU holding the lock.

If the CB was not in the TCB of the owning PU, the CB is not recovered by the recovery scheduled for that PU. Therefore, this CB has inadvertently been left locked and is a possible cause for the hang. Hang recovery then "steals" the lock by overwriting this lock (by means of a compare and swap instruction [8]) with the lock identity of the PU that was hung. This CB is then recovered during this hung PU IOSS CB recovery task in the same fashion as other CBs locked by this hung PU.

If it turns out that the PU holding the lock of the CB is a running PU that is not scheduled for recovery, additional steps are taken to ensure that a consistent state exists between the CB lock and the TCB of the PU holding the lock before determining the recovery action to take. For example, if the CB was being locked and unlocked by a PU in either a temporary or an endless code loop, the lock would be in a state of transition. Hang recovery could not steal the lock in this case, because this transition could result in the lock being overlaid back to the running PU in the code loop. Therefore, the action for this hang recovery is simply to remove this CB from the TCB of the hung PU. For this specific case, the concept is as follows: If the running PU is in an endless loop locking and unlocking a CB, the watchdog timer for that PU eventually indicates a timeout condition, which causes that CB to be recovered by a subsequent hang recovery task for that PU.

If the lock of the CB held by the running PU is not in transition, the TCB of the running PU can be searched for this CB. If the CB appeared in the TCB of the running PU, we can be reasonably certain that the running PU knows about the lock. Therefore, the action for the hang recovery currently being processed is simply to remove this CB from the TCB of the hung PU. Even if the PU holding this lock is hung performing some other action that caused this CB to be left locked, hang recovery for that PU should initiate at some subsequent point and recover the locked CBs in its TCB, thereby recovering the CB that caused the initial hang.

However, if the CB does not appear in the TCB of the running PU and the lock is not in a state of transition, it is this PU that is inadvertently holding the lock that left the CB locked. Hang recovery "steals" the lock by overwriting this lock (by means of a compare and swap instruction [8]) with the lock identity of the PU that was hung. This CB is then recovered during the hung PU IOSS CB recovery task in the same fashion as other CBs locked by this PU. It is interesting that for this case, the CB can be "stolen" and recovered without putting the running PU that caused the hang through any explicit recovery actions. This scenario is depicted in **Figure 3(b).**

In a later section of this paper, testing is discussed. Analysis of FFDC data from this testing revealed that the following cases were injected and the subsequent recovery actions were successful. The hung CB was locked by a PU that either was being recovered—in which case the hung CB address was removed from the TCB of the hung PU—or was not being recovered—in which case the hung CB followed the algorithm in Figure 3(b).

Parallel recovery considerations for hang recovery

Prior to the advent of the z9, designs for parallel recovery had been attempted with little success. The main obstacle to achieving a workable design was in the area of IOSS CB recovery. Here too, the scanning for CBs that were left locked by the PUs that were to be recovered was also problematic. For a parallel recovery design to be of any performance benefit, the scanning of the CBs would have to be divided among the SAPs dispatched to perform the recovery in parallel.

With the new TCB infrastructure on the z9, the split of which CBs to recover aligns very nicely with the ability to dispatch multiple SAPs to perform recovery in parallel, whereby each SAP is assigned a non-overlapping subset of the PUs to recover. Because the TCBs are organized on a PU basis, with each TCB containing CBs either locked or attempting to be locked by that PU, it simplifies the determination of which CBs each SAP is to recover. Each SAP performing recovery now examines the TCBs of the PU being recovered to determine which CBs to recover.

In the situations mentioned above, there are cases in which CBs could be listed in multiple TCBs. The methods for hang recovery were designed with parallel recovery in mind to resolve potential overlap situations. Whether or not there is a hang, just prior to IOSS CB recovery a portion of the hang recovery firmware is run that resolves any TCB CB overlap. This overlap is eliminated by ensuring that CBAs left in the TCBs that are undergoing IOSS CB recovery are the only valid CBAs for CBs locked to the PUs being recovered by the SAP. Hence, after any overlap is resolved, the TCBs of the PUs being recovered contain only valid addresses of CBs that are locked to the PUs holding the CB lock.

To avoid having to lock TCBs for PUs that other SAPs are recovering, hang recovery employs methods a SAP can use to "steal" the CB lock if it is required, as shown in Figure 3(b), rather than inserting CB information into a TCB for a PU the SAP is not recovering. Not having to lock the TCBs reduces contention in a parallel recovery environment.

Parallel recovery execution

In System z servers prior to z9, the FFDC data collection for multiple errors in the I/O hardware components and

the recovery actions for the self-timed interface (STI) network [2] were already executed in parallel by multiple processors. During this phase of recovery, called *hardware error recovery*, all error indications in the I/O hardware are reset, the resulting IOSS recovery actions are determined, and appropriate requests are put into an order-book data structure called the *global recovery request block* (GRCB), which is visible to all SAPs in the system. After completion of these tasks, the recovery actions related to IOSS control blocks and channel reinitialization are performed sequentially by the affected SAPs on the basis of the requests in the GRCB. While reinitialization of channel hardware could already have been done in parallel on all SAPs without difficulty prior to z9, sequential execution was required to restore consistency and recover the IOSS control blocks because of the structure of the algorithms used in this phase. This restriction was removed on z9 by exploiting the new TCB infrastructure that was described in detail in the previous section.

The only remaining synchronization point is between I/O hardware recovery and IOSS CB and channel recovery. On one hand, this is due to the hierarchical structure of I/O hardware. For example, hardware problems present in the STI network must be recovered before problems in attached components can be recognized and handled. On future systems, it may be an evolutionary step to enhance the granularity of this synchronization by, for example, performing recovery for channels in one domain of the STI network even though there are still errors pending in other domains. However, at present, the latest recovery improvements are more than sufficient to keep pace with the overall growth in system size and complexity. The additional effort in state tracking and bookkeeping of failure details would require a highly complex infrastructure that, by itself, would create an exposure to new failure scenarios.

The I/O hardware error-handling phase of recovery is relatively fast, usually taking less than 100 milliseconds. In contrast, the IOSS control-block-related recovery and the channel reinitialization based on the requests collected in the GRCB may take several hundred milliseconds. Since reinitializing the channels requires very little processing capacity on the SAPs, handling multiple channels in parallel takes only slightly longer than reinitializing a single one. When multiple I/O hardware errors occur at about the same time, recovery on the z9 exploits this performance benefit by executing this phase of the recovery only after waiting until the requests from all I/O hardware errors pending in the system have been collected in the GRCB.

To support parallel IOSS recovery, the z9 introduces improvements in the error reporting to the recovery and the bookkeeping for pending recovery actions. On previous systems, each SAP scanned the various reporting interfaces used by other SAPs to move orders into a common system-wide order book (global recovery CB or GRCB) before waiting at the synchronization point. This approach can be thought of as a *pull concept*. Once the I/O hardware recovery comes to an end, this SAP continues its work by picking all orders from the order book that can be executed on this SAP. New requests that are pending on the reporting interfaces are not picked up by the current recovery run, but are left for a later recovery run.

With z9, each PU reporting an error can place a recovery request in the GRCB directly and make the order known to any SAP that is waiting at the synchronization point—a *push concept*. This strategy guarantees a more complete view of the system state while performing IOSS recovery. New orders targeted for an SAP that is already waiting at the synchronization point can be executed as soon as possible. As a result of these improvements, the z9 is the first System z mainframe that allows full parallel execution of IOSS recovery.

Testing enhanced recovery

During z9 enhanced recovery testing, special error-injection tools and techniques were used to emulate various PU hardware failures at various firmware code points. The tool uses a software interface to the processor hardware to force uncorrectable errors at specific code points in millicode to invoke recovery. These code points were both focused and randomized to ensure that the emulated PU failures occurred after various CBs were left locked by one or more of the failing PUs. The error injections were done in various z9 environments—from small z9 central electronic complex simulator (CECSIM) [10] environments to huge z9 machine configurations with many processors and large I/O configurations. In many cases, the error injections were done on the z9 while the system was under a heavy load running various test programs in various LPAR partitions. In addition, firmware errors, such as setting locks into various CBs to emulate CB hang situations, were done in these various test environments as well.

Several immediate benefits were seen in the enhanced recovery testing. The sympathy sickness often seen on systems prior to z9 with large-system configurations during system recovery was not seen in testing. This was primarily because of the increased speed at which recovery runs with the new design. Also, several code bugs were found during testing that inadvertently left a CB locked by a running processor that, on systems before z9, would have caused a system check-stop. The z9 successfully recovered from these errors, thereby validating the z9 CB hang recovery design.

The enhanced recovery improvements also proved beneficial in the overall debugging of other z9 functions that were under test:

- Code bugs found during the normal testing process for new firmware were identified closer to the point of origin because of the advanced error detection.
- Added state tracking identified the task or instruction executing at the time of error.
- Improvements related to enhanced recovery that were made to the FFDC logs and traces resulted in the precise logging of the CBs that were recovered or caused the CB hang.
- The precise error codes that were added to the TCB allowed for faster identification of bugs.

Conclusion

The z9 TCB infrastructure and its supporting firmware have improved the IOSS recovery function by eliminating lengthy CB scans, by providing enhanced error detection and recovery for firmware-related problems, and by providing a solution that scales with the ever-increasing demands placed on the I/O subsystem by enterprise-level computing.

References

1. L. W. Wyman, H. M. Yudenfriend, J. S. Trotter, and K. J. Oakes, "Multiple-Logical-Channel Subsystems: Increasing zSeries I/O Scalability and Connectivity," *IBM J. Res. & Dev.* **48**, No. 3/4, 489–505 (2004).
2. D. J. Stigliani, Jr., T. E. Bubb, D. F. Casper, J. H. Chin, S. G. Glassen, J. M. Hoke, V. A. Minassian, J. H. Quick, and C. H. Whitehead, "IBM eServer z900 I/O Subsystem," *IBM J. Res. & Dev.* **46**, No. 4/5, 421–445 (2002).
3. 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).
4. L. C. Alves, M. L. Fair, P. J. Meaney, C. L. Chen, W. J. Clarke, G. C. Wellwood, N. E. Weber, I. N. Modi, B. K. Tolan, and F. Freier, "RAS Design for the IBM eServer z900," *IBM J. Res. & Dev.* **46**, No. 4/5, 503–521 (2002).
5. M. L. Fair, C. R. Conklin, S. B. Swaney, P. J. Meaney, W. J. Clarke, L. C. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber, "Reliability, Availability, and Serviceability (RAS) of the IBM eServer z990," *IBM J. Res. & Dev.* **48**, No. 3/4, 519–534 (2004).
6. J. Maergner and H. R. Schwermer, "High Level Microprogramming in i370," *The Design of a Microprocessor*, W. G. Spruth, Editor, Springer-Verlag, New York, 1989, pp. 303–316.
7. L. Spainhower and T. A. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM J. Res. & Dev.* **43**, No. 5/6, 863–873 (1999).
8. IBM Corporation, *z/Architecture Principles of Operation,* Document No. SA22-7832-04; see *http://publibz.boulder.ibm.com/epubs/pdf/a2278324.pdf.*
9. L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries Processor," *IBM J. Res. & Dev.* **48**, No. 3/4, 425–434 (2004).
10. J. von Buttlar, H. Böhm, R. Ernst, A. Horsch, A. Kohler, H. Schein, M. Stetter, and K. Theurich, "z/CECSIM: An Efficient and Comprehensive Microcode Simulator for the IBM eServer z900," *IBM J. Res. & Dev.* **46**, No. 4/5, 607–615 (2002).

Received March 10, 2006; accepted for publication July 11, 2006; Internet publication January 12, 2007

Kenneth J. Oakes *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (oakes@us.ibm.com).* Mr. Oakes is a Senior Technical Staff Member in eServer I/O development. He received a B.S. degree in electrical engineering from the University of New Haven. He is currently involved in IOSS design for the next-generation System z eServer. Mr. Oakes holds numerous patents and awards relating to channel and IOSS design.

Ulrich Helmich *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (helmich@de.ibm.com).* Mr. Helmich received an M.S. degree in physics from the University of Sussex and a Dipl. Phys. degree from the University of Tuebingen, Germany. His current responsibilities involve zSeries I/O microcode development, including first-error data collection (FEDC) and error-recovery code.

Andreas Kohler *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (akohler@de.ibm.com).* Dr. Kohler received M.S. and Ph.D. degrees in physics from the University of Stuttgart, Germany. His current responsibilities involve System z I/O firmware development, including test tools, simulation, and error-recovery code.

Andrew W. Piechowski *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (fireman@us.ibm.com).* Mr. Piechowski is a Senior Software Engineer in System z9 coupling firmware development. He received an A.S. degree in electrical engineering from Thames Valley State Technical College. Mr. Piechowski has received several IBM Outstanding Technical Achievement Awards for his work on microcode design and development.

Martin Taubert *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (taubert@de.ibm.com).* Mr. Taubert is an Advisory Engineer in System z firmware development. He received his M.S. degree in computer science (Dipl.-Inform.) from the University of Kaiserslautern, Germany. He has received two IBM Outstanding Technical Achievement Awards for his contributions in the recovery area. Mr. Taubert is currently involved in IOSS design for the next-generation System z eServer.

John S. Trotter *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (trotterj@us.ibm.com).* Mr. Trotter is a Senior Software Engineer. He received his B.S. degree in electrical engineering from the Polytechnic University. He is currently involved in IOSS design and development for the next-generation System z eServer. Mr. Trotter holds numerous patents and has received several awards, including an IBM Outstanding Technical Achievement Award for the development and testing of the multiple image facility (MIF).

Joachim von Buttlar *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (joachim_von_buttlar@de.ibm.com).* Mr. von Buttlar is an Advisory Engineer in System z9 firmware development. He received an M.S. degree in computer science (Dipl.-Inform.) from the Technical University of Berlin, Germany. His current responsibilities include development of z/CECSIM and IOSS design for the next-generation System z eServer. Mr. von Buttlar has received an IBM Corporate Award.

Robert M. Whalen, Jr. *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (rmwhalen@us.ibm.com).* Mr. Whalen is a Senior Software Engineer. He received a B.A. degree in computer science from the State University of New York at Potsdam. He has worked in both the hardware and software laboratories on many projects as a microcode and software developer. Mr. Whalen has received several IBM Outstanding Technical Achievement Awards for his work on microcode design and development.