Functional verification of the z990 superscalar, multibook microprocessor complex

D. G. Bair
S. M. German
W. D. Wollyung
E. J. Kaminski, Jr.
J. Schafer
M. P. Mullen
W. J. Lewis
R. Wisniewski
J. Walter
S. Mittermaier
V. Vokhshoori
R. J. Adkins
M. Halas
T. Ruane
U. Hahn

This paper describes the verification methods and techniques that were established to verify the microarchitecture and architectural correctness of the z990 microprocessor and storage subsystem. The ring-based, four-book storage subsystem links 64 superscalar microprocessors together in this system. The verification process started at the unit level, which focused on the correctness of the microarchitecture, and then proceeded to the element level to verify the architectural correctness of the microprocessor and storage subsystem. After successfully completing element stress testing, the components were combined and verified at the system level. Since the methods used at system-level verification were much the same as the ones used on the CMOS-based IBM S/390® Parallel Enterprise Server G4, the focus of this paper is on the work done at the unit and element levels.

Introduction

A major logic verification effort was required to verify the functionality of the z990 superscalar, multibook microprocessor complex. The verification team developed and executed a staged verification plan based on a broad range of verification technologies and collaborated with the logic design team to formulate the verification plan to ensure that all features of this complex system were fully verified.

To successfully verify this design point, the verification team built on the hierarchical verification approach used on the CMOS S/390* Parallel Enterprise Server G4 System [1]. For this system, the verification effort began at the protocol level using an abstract model. It then continued at the designer level, subunit level, unit level, multiunit level, element/chip level, and finally system level. The abstract protocol model, using a formal verification tool called the Murphi Verifier, allowed the team to verify the multibook protocol before the design was committed to a hardware description language (HDL). All major units in the microprocessor were first verified in a unit simulation environment. Subunit and multiunit simulation environments were created as necessary to fully stress the microarchitecture.

The verification team included a core of expert verification engineers who were responsible for setting the direction of the different simulation environments. These experts would select the simulation logic boundaries as well as the verification technology used to verify a particular piece of the design point. Both proven verification technologies and new, leading-edge techniques were used to verify this complex design point. There was no set formula to determine which technology to use for a given environment. The verification techniques from which the team of experts selected included the following technologies: Murphi Verifier, extended coverage models, asynchronous clock modeling [2], random command-driven techniques, architectural test-case generators, system reset verification, disabled or degraded configuration verification, recovery verification, and performance verification techniques.

0018-8646/04/$5.00 © 2004 IBM

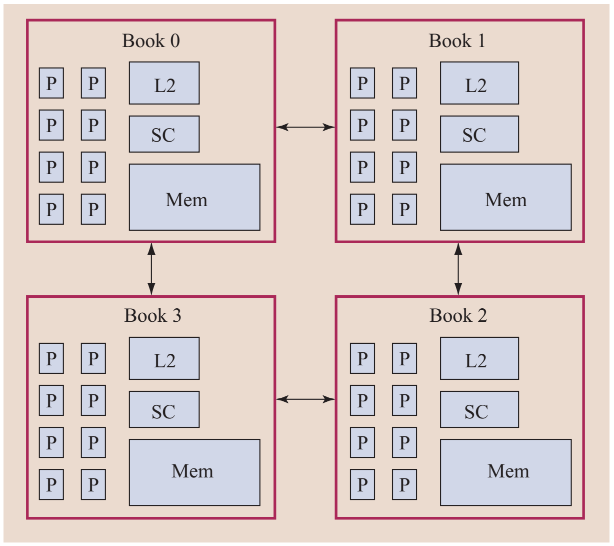


Figure 1

Four-book z990 system. (P: processor; Mem: section of main memory; L2: L2 cache; SC: system control.)

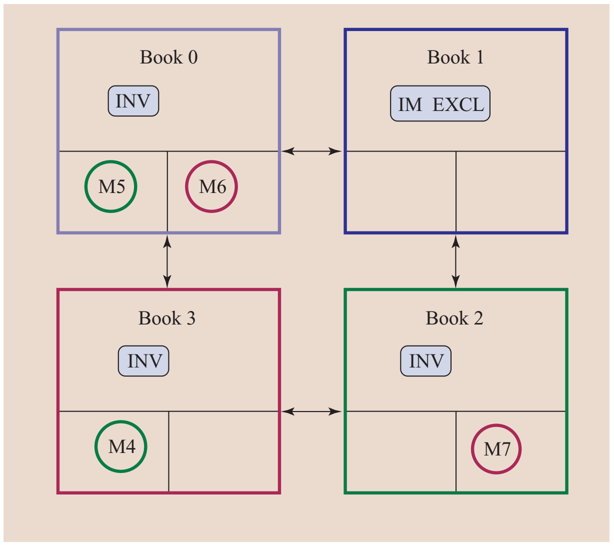


Figure 2

Four-book z990 system showing a typical state of the protocol model. M4 and M5 are messages issued by Node 2; M6 and M7 are messages issued by Node 3. The three cache-line states are EXCL (exclusive); IM (intervention master); and INV (invalid).

Formal verification of the memory ring protocol

Figure 1 illustrates a four-book z990 system. Each book contains processors (P), a section of main memory (Mem), an L2 cache (L2), and a system control (SC). The books send and receive messages on a bidirectional ring. When a processor issues a memory request that cannot be satisfied locally within the book, the SC sends messages to the other books on the ring. The messages that flow on the ring and the rules for processing these messages form the *memory ring protocol*.

Because the memory ring protocol for the system was a new design with many novel features, the design team considered it important to formally verify the protocol. Over the course of the project, the formal verification process revealed a number of crucial problems, which led to major changes in the design of the protocol. Significantly, most of these problems were not detected in the course of random simulation of the logic-level implementation.

The general method for analyzing a complex protocol, such as the z990 ring protocol, is to create an abstract *protocol model* that can be checked by systematically examining all of its reachable states. There are two reasons why the team needed to use an abstract model instead of attempting to formally verify the logic-level implementation of the protocol. First, the logic-level implementation was much too large to formally verify by current methods. To check the correctness of the z990 ring protocol, the team had to show that when the four books interacted on the ring, no errors could arise from the interaction. While the logic-level implementation of the four books was much too large to verify formally, a protocol model of the four books was small enough to be fully analyzed. The second reason for building a protocol model was that it allowed the designers to check the correctness of new protocol features before implementing them. The team found that this ability saved them valuable time in the design project.

The tool used by the team to model and verify the protocol was a version of the Murphi Verifier [3], extended to increase its capacity for verifying large systems. The extended verifier used data stored on disk as well as in main memory [4].

The team constructed a protocol model that represented the operation of the four books with respect to a single memory address. The purpose of the model was to check whether the protocol maintains a few crucial memory system properties, including *coherence,* across the four books. With the property of coherence, if one book has a copy of the data marked *exclusive,* no other book has a copy of that data. Since there was little interaction between memory operations on different addresses, properties such as coherence could be analyzed in an abstract model having only a single memory address.

A typical state of the model is illustrated in **Figure 2**. The state of each book is represented by a box, where the upper part of the box contains information about the directory state of the book and the lower part of the box shows which messages are present at the book. The left and right halves of the message area correspond to different stages of message processing within the book. In the ring protocol, when a book issues a memory request, it sends two messages around the ring in opposite directions. The figure shows two messages from the red book (Book 3) and two messages from the green book (Book 2). The actual model contained many state variables not shown in the diagram.

Our protocol modeling and verification effort started in 2000 and continued for two years. During this time, the protocol was in constant revision, partly in response to problems detected by protocol verification and partly owing to problems detected by other traditional simulation environments. One advantage of the team's approach was that the model could be changed very rapidly and reverified as the protocol evolved.

The types of errors found by protocol verification included

- Errors in the complex rules governing when messages can be issued on the ring and the order in which messages are processed in a book.
- Errors in the rules by which books keep track of the operations that are currently pending (in progress).
- Errors in the rules for assigning accept or reject responses to memory requests.

During development, random simulation was applied to the system implementation continuously. In some cases, an error found by protocol verification remained uncorrected in the implementation for many months. It is interesting to note that random simulation did not detect these errors, even after months of searching with simulation.

As a result of the experience of using protocol verification on the z990 system, the Systems and Technology Group is planning to use formal protocol verification in its future projects. The team found that the ability to check protocol features prior to implementation saved valuable time in the design process.

Memory storage controller verification

The z990 was the first zSeries* server with a memory storage controller (MSC) unit-level simulation environment. This environment was developed as a result of exposures found during the initial power-on of previous design points. The analysis of test-floor escape for prior systems indicated a weakness in the ability to stress the MSC design with various memory configurations and a need to narrow the focus of the initial power-on sequence.

The MSC verification environment was built on the same platform as the SC environment, described in more detail in the next section. There were two main reasons

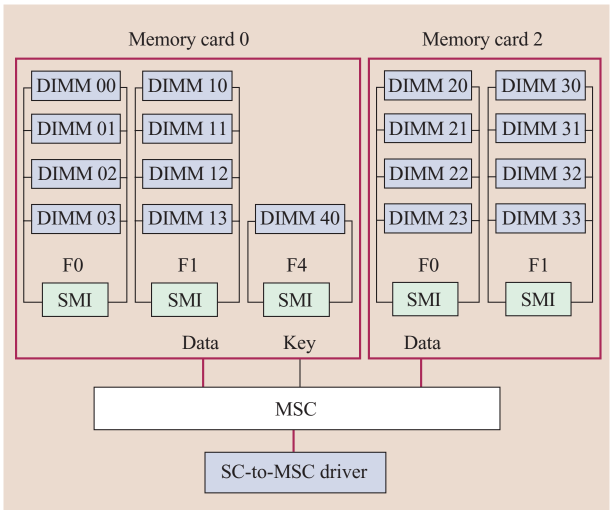


Figure 3

Denali MMAV-based memory model.

for this. The first was to share as much of the verification environment as possible to avoid duplication of common sharable resources. This allowed us to use a very small team to control and maintain a completely separate environment. Second, this allowed the two environments to be seamlessly joined into a larger test platform with minimal changes to the environment. This larger environment also provided a feedback process for the MSC unit platform. All problems that were uniquely found in either SC or system simulation models were analyzed, and enhancements were made in the MSC unit environment to eliminate this exposure in the unit model.

Prior test-floor feedback also indicated that the team needed to enhance the verification of the memory interface. During our analysis, we investigated tools that were available in the marketplace to aid in the testing of the memory design. We discovered that Denali Software produced a tool called Denali MMAV** that verified DRAM memory specifications. We were able to retrofit the Denali MMAV tool into the MSC environment with the help of our internal simulation tools department. With this tool, we were able to enhance the checking on the memory interface.

The Denali MMAV-based model is shown in **Figure 3**. The main purpose of this model was to verify the protocol between the MSC design and a virtual representation of the main memory cards. The connection to memory was accomplished via the synchronous memory interface (SMI). The SMI is a separate chip that provides a connection to the real memory. In this model, we used the SMI simply to provide a path to the memory. The Denali MMAV tool was designed around dynamic random-access

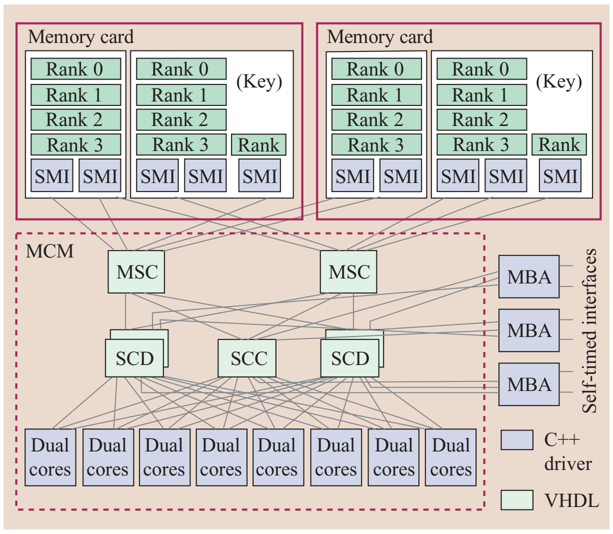


Figure 4

One-book system control simulation environment.

memory (DRAM) technology. Our memory design used industry-standard dual inline memory module (DIMM) technology. To use the MMAV tool, the user needed to build a model of the DIMM, including any necessary buffers. Building a strict representation of memory would have taken 72 DRAMs per SMI chip (4 DIMMs × 18 DRAMs/DIMM) and a large amount of overhead in the complicated modeling connections. Our goal here was not to verify the DIMM, but to prove that the MSC design was capable of functioning with the DIMM technology. To simplify the connection to memory, we used the MMAV tool to widen the bit definition of the DRAM data to build our minimal configuration. By evaluating the memory connections, we removed the redundant DIMM connections to provide this simplified representation. This simplification also provided a cleaner memory layout for memory initialization. Our new model took a 72-DRAMs-per-SMI chip to a minimum of four "super" DRAMs per SMI chip. Using this simplified memory connection, we were able to issue all memory commands through the Denali MMAV tool and verify the MSC-to-main-memory protocol.

The Denali MMAV tool did not come without a financial burden; every simulation run required a license from Denali Software. We needed to balance the value added by using the Denali model against the cost of the licenses. Since the Denali environment was almost as transparent as moving to the SC environment, we felt that we needed only to spot check some percentage of tests to prove that the base MSC unit platform was not violating any memory specifications. The team decided that ten Denali MMAV licenses would give us sufficient capacity while limiting the cost.

SC verification

The SC verification environment focused on testing the functionality of the system controller control (SCC) and system controller data (SCD) chips, although it also aided in the testing of the MSC chip because the MSC hardware description language (HDL) was also contained in the SC simulation models. However, as described in the previous section, the MSC was more thoroughly tested in its own environment.

The SC provides four basic functions in the z990 system:

1. It interconnects four books of eight dual-core processor chips via a ring fabric.
2. It provides a second-level shared cache between the private L1 cache in each processor and the memory cards.
3. It acts as a cache for certain input/output (I/O) data accesses by the memory bus adapter (MBA).
4. It provides data management functions for data sharing between processors and I/O, maintaining data coherency, and guaranteeing that each processor is always working with the most recent copy of data.

Two main simulation models were used in the SC verification environment: a one-book model, shown in **Figure 4**, and a four-book model. The four-book model could be dynamically configured at run time to represent any supported system configuration (e.g., two-book closed ring, three-book open ring, and so on), which allowed us to avoid having to build and maintain a multitude of models. The four-book model could even be configured as a one-book system, but we kept the separate one-book model since the model size was smaller and the simulation jobs ran faster.

SC chip-level verification was accomplished by building on "random-based" environments and methodologies used on previous S/390 systems, such as the G4 system [1]. However, the z990 SC had many new design features, such as the ring fabric, that posed significant new challenges to the verification team.

One significant improvement to the SC verification environment was to add "looping" capabilities to the processor driver. From the team's experience with previous systems, we recognized that one weak point of a random-driven environment is that it is inherently difficult to generate synchronous-type loops, also called *livelocks*. In the past, the team had test-floor escapes in which a bug in the SCC could cause a processor request to "starve" and ultimately hang. This class of bug would be uncovered only by creating a repeated pattern of commands; any slight deviation from the pattern would cause the errant logic to break out of its livelock state. The team implemented two different loop modes in the processor driver—a *user loop* mode and a *random loop* mode. The user loop mode gave the user the ability to create a specific set of processor commands, including cycle delays, on which the processor driver would loop during execution. The random loop mode allowed the user to specify a list of commands, and the processor driver would choose its own loop using those commands. These loop modes actually enabled the team to catch several SCC bugs that would otherwise have escaped to the test floor.

The move to four books from the two nodes of previous systems introduced a new ring structure and coherency protocol. Owing to the ongoing evolution of the ring design, no stimulus was directly applied to the ring interface because of changes in the interface protocol. Ring traffic was generated by driving the microprocessor and I/O interfaces with commands that would, in turn, cause the SC to generate commands across the ring. The team took measurements of bus utilization and determined that sufficient ring traffic was being generated and a separate ring driver was not required.

The ring protocol depended on the following states: *intervention master* (IM), *memory master*, *multicopy*, and *local directory* state information. Different combinations of directory states were warm-loaded into the SC arrays at the beginning of the simulation runs. This allowed the test cases to reach interesting directory states early in the simulation run. A set of remote access bits were also used in the configuration array segment boundaries to reduce ring traffic. This was needed for addresses being used both locally and exclusively. These bits were preloaded for coherency and to influence the ring traffic.

To correlate the Murphi abstract model with the actual model, the rules from the Murphi model were implemented as checks on the HDL model. Even with this effort, differences between the Murphi and HDL models existed. These differences were exposed when a number of problems were found on the test floor. The checking drawn from the Murphi model was expanded after a number of coherency problems were found in the hardware. These problems involved long delays that resulted in defeating the IM bit address protection and resulted in out-of-order processing of ring requests. The team should have introduced longer delays affecting individual requests in the verification process, either by direct manipulation of internal wires or latch values in the simulation model or by using some of the disable switches that were included in the hardware. This would have uncovered many of the test-floor escapes that resulted from long delays and for which the blocking conditions for large blocks of time were rare.

The team collected statistics on the ring priority logic (requestors granted during request period, number of cycles before grant in request period). While we initially collected this information in order to assess efficiency, we later used and augmented the statistics to fine-tune the priority to correct problems discovered on the test floor. The improvements to the statistics involved increasing the categories of requestors so that some categories could be ignored while others were watched more closely.

Another new feature of the ring structure was the ability to perform concurrent book upgrades in which new books with processors, I/O, and memory could be added without having to take the machine offline. The testing of this feature was implemented with a control sequence located primarily in the processor driver, where the millicode operations for the sequence were generated and sent. The testing of the upgrade function involved quiescing the processor (see the section on quiesce below), holding I/O, and changing ring fences as needed to open the ring, add a book, and close the ring. While the ring fences were active, random patterns with and without good error-correction code and parity and interface alignment protocol (IAP) patterns were directed against the fenced ring interfaces. As long as the ring fences were verified in this environment, the team did not think it necessary to run the actual IAP sequence while performing the concurrent book upgrades. The IAP sequence was tested separately in another simulation environment, saving thousands of cycles in the concurrent book upgrade sequence. When a new book was brought online, additional addresses were made available to the processor and I/O drivers that resided in physical memory on the new book.

Elastic interfaces (EIs) were used on the SC–MBA interface, book-to-book interface, and MSC–SMI interface. The EI logic was responsible for presenting all of the incoming interface data to the receiving I/O blocks on the same clock cycle, even though the interface lines could have been skewed over a cycle boundary. The cycle in which the EI logic presented data to the receiving I/O blocks was known as the *target cycle*. The logic for the EI was verified separately using gate-level models before the logic was tested at the element level. The goal for verification at the element level was to verify the operation of the IAP controls, the interface fences, and the effects of changes in delays due to wires and EI target cycles. Verification of the code that drives the IAP process was done at the system level.

To take advantage of the faster one-cycle simulation models, the team replaced the EI logic in those models with dummy HDL that modeled the delays for the EI target cycle. The team modeled wire delay using a newly developed bidirectional HDL wire delay driver, which was added to the simulation model during the model-build

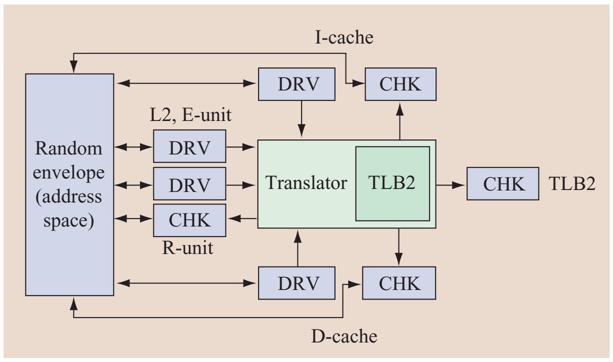


Figure 5

Translator unit simulation environment. (E-unit: execution unit; R-unit: register unit.)

process without affecting the HDL design libraries. This form of modeling the wire delay was used in element simulation and also in system simulation. Because it was an HDL solution, it could be used on the hardware accelerators during system simulation initial machine load testing.

Care was taken to correlate the target cycle values used in the one-cycle and two-cycle models [1, 5] with EI logic and the hardware running on the test floor. Because IAP took up to 200,000 simulation cycles for some interfaces, a post-IAP state was captured after completing IAP so that it could be loaded into the model without running IAP for the majority of the two-cycle test cases. These states were affected by the wire delay used, so a number of states were collected and the appropriate one was loaded with respect to the wire delay being used in a particular simulation model. The same post-IAP states were used in element and system simulation. By using the actual post-IAP states, the EI target cycle settings remained consistent in all environments, whether running our simulation models (either through the actual IAP sequence or after loading a post-IAP state), or running real hardware on the test floor.

The ring IAP presented an extra level of complexity because five chip-to-chip connections were needed to connect one book to another on each ring port. One SCC and four SCD chips on each book had to communicate with their counterparts on the other book. Because the IAP calibrated an interface to the worst delay encountered on the wires for that interface on a per-chip basis, and because all five chips had to see the same delay for the logic to function correctly, an extra *voting* step was used for the ring calibration in which the worst case was determined for the five chips and shared with the other four chips. Voting was followed by another step to ensure that the calibration values were correct. The team programmed the wire-delay driver to show different delays between the five chips in order to verify that the voting function worked correctly.

Initially, delays between pairs of books were set over a random range centered on the expected nominal delays for the book interconnects across the backplane. The logic was not designed for completely arbitrary delays around the ring, and adjustments were made to constrain the delays to more closely reflect the physical placement of cards in the backplane and, where a book was not installed, the use of shunting cards. A range was still used to allow for differences in cycle times, but the delays across each leg were now related to the delays on the other legs. While some differences were anticipated because of placement on the backplane and because variance was added in simulation to account for this, there was another source of delay differences. When cycle times cause the delays to be near a cycle boundary, the delay across the wire could just make or just miss timing, thereby causing a deviation from the balanced delays, with one or more legs seeing an extra cycle. The design was not expected to handle this case, and the EI was used to balance the delays by adding additional cycles to the legs that fell on the short side of the delay boundary under service processor control.

IAP is controlled through a set of registers that is accessed via the clock serial interface. Using the serial interface in the SC model would have required too many simulation cycles, so software drivers were used to modify the latches directly. Because the serial interface timing between chips could vary, variable delays for the model accesses between chips were added to identify any problems that might result from the timing differences.

Processor verification

The verification of the processors in a z990 system—because of their size and complexity—was done at both the unit and the processor level. Unit environments were created for the translator unit, the instruction unit (I-unit), the execution unit (E-unit), and the buffer control element (BCE) unit. In addition, a subunit environment was created for the operand buffer. The structure of a unit simulation environment is shown in **Figure 5**.

In the past, the BCE was the only unit that had an extensive unit environment. The team added a unit environment for the translator because its logic was now in a separate unit (it had previously been part of the BCE) and, with the addition of a second-level translation-lookaside buffer (TLB2) (discussed below), the translator was now more complex than on previous machines. Extensive I-unit and E-unit environments were created because of both past test-floor escapes and the increased complexity of the z990 design. Finally, the team created a subunit environment for the operand buffer because, in addition to its complexity, there was no way to adequately test this function at the element level. The team found that by concentrating on verification at the unit level, problems could be discovered and fixed more quickly and that the resulting element simulation environment ran with fewer errors than on previous machines.

Translator unit verification
The zSeries address translator translates virtual addresses used by programs into absolute addresses used to access real storage. The translation can involve several table lookups (up to 35) as well as a prefixing operation (done to relocate the first 4K of storage). The address translator must be able to support 24-, 31-, and 64-bit addresses. To improve performance, the translator contains a TLB2, where information on recently translated addresses is kept.

The team used an address translation reference model called *address space* to provide us with a set of virtual addresses and the data required for the translation process. The address space and other generic functions (for example, a random number generator) used for simulation were contained in a shared library called the *random simulation environment*. Addresses were randomly picked from the address space by software drivers and used to stimulate the inputs of the translator. Software monitors were connected to each interface to check protocol conformance and expected results.

In addition to the interface checkers doing a black-box check, we also used a white-box checker for the TLB2. This was needed to verify performance characteristics of the TLB2 that were not visible at the interfaces of the translator. The TLB2 checker was a software representation of the real hardware TLB2. To avoid simulation speed slowdown, the equivalence between hardware and software was checked only when the TLB2 contents changed and when the test case ended. The TLB2 checker was also used to preload the TLB2 and to ensure a graceful degradation of the TLB2 arrays.

Operand buffer subunit verification
The random test-case driver approach was used to verify the functionality of the operand buffer control logic [1]. As shown in **Figure 6**, the environment included the operand buffer controls, dataflow logic, and fetch control logic. These were surrounded by drivers representing the I-unit (where address generation was done), the E-unit, and the BCE, as well as by software monitors.

The monitors/drivers were responsible for initiating requests, responding to logic requests, initiating cross-invalidates (XIs), and verifying the results at the end of each request by checking the correctness of the data and exceptions being reported. All driver operations were under the control of various parameter files. This allowed control of the delay between request and responses, XIs, exceptions, branch wrong indications, and BCE address compare, allowing specific areas to be easily targeted.

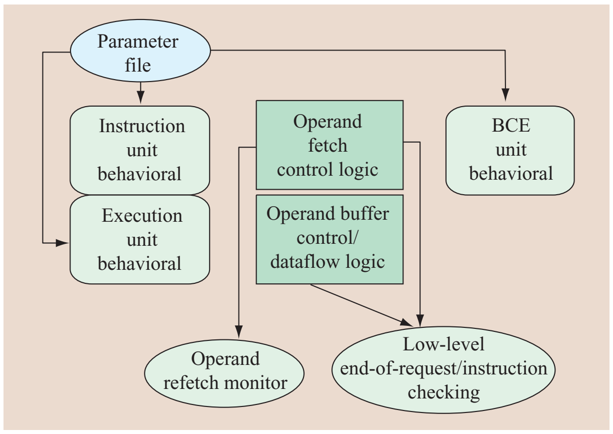

Parameter file
Instruction unit behavioral
Execution unit behavioral
Operand fetch control logic
Operand buffer control/ dataflow logic
BCE unit behavioral
Operand refetch monitor
Low-level end-of-request/instruction checking

Figure 6

Translator unit simulation environment.

It was particularly difficult to verify the operand refetch mechanism. This was complicated by the out-of-order data returned from the BCE. The team verified this by modeling the operand buffers in software, keeping track of valid operand buffers as well as the addresses and data associated with each operand buffer. Simulation code would randomly pick a cycle to force XIs, obtain an address to invalidate, and force XIs to all operand buffers associated with that address. At that time, the actual buffer data was modified to cause a failure if old data was used before refetch and the first buffer to be refetched was saved. As each request was processed, the code verified that the correct buffers were refetched at the correct time. This mechanism allowed the buffers to be refetched multiple times per instruction. Because the data could be returned out of order, the monitors had to ensure that only the correct buffers were refetched and that no buffer with old data was used. Using this method, the team was able to quickly discover and fix complex problems at the unit level.

I-unit verification
The methodology used in the I-unit verification environment was the random test-case driver approach described in [1]. This allowed for much longer test cases to be run in a shorter period of time, since there was no need to generate a test case prior to run time. Another reason for using the random test-case driver method for unit simulation was to differentiate it from the processor element environment, which used pregenerated test cases. The different methodologies helped to expand the testing

```
//-------------------------------------------------------------------------------------------------------------
// opcode enab excp fmt arc exe  sup grp pip fwd zin dov zck p1c sto fet r1   r2   ru  mis alg  mis2  mnem
//----------------------------------------------------------------------------------------------------------
      {  0x38,  10,  - ,  1,  1,  1,  5,  3,  6,  0,  4,  4,  9,  0,  0,  0,  55,  51,  0,  0,  0,  0,  "LER"  },
      {  0x39,  10,  - ,  1,  1,  1,  5,  3,  6,  0,  4,  4,  9,  0,  0,  0,  51,  51,  0,  0,  0,  1,  "CER"  },
      {  0x3B,  10,  - ,  1,  1,  1,  5,  3,  6,  0,  4,  4,  9,  0,  0,  0,  58,  51,  0,  0,  0,  1,  "SER"  },
      {  0x3F,  50,  - ,  1,  1,  1,  5,  3,  6,  0,  4,  4,  9,  0,  0,  0,  58,  51,  0,  0,  0,  1,  "SUR"  },
      {  0x40,  50,  - ,  2,  1,  1,  1,  5,  3,  4,  2,  4,  9,  1,  1,  0,  11,  00,  0,  0,  0,  0,  "STH"  },
      {  0x41, 250,  - ,  2,  1,  1,  1,  5,  3,  3,  4,  4,  9,  1,  0,  0,  45,  00,  0,  0,  0,  0,  "LA"   },
      {  0x42,  50,  - ,  2,  1,  1,  1,  5,  3,  4,  2,  4,  9,  1,  1,  0,  11,  00,  0,  0,  0,  0,  "STC"  },
      {  0x43,  50,  - ,  2,  1,  1,  1,  5,  3,  4,  1,  4,  9,  1,  0,  1,  18,  00,  0,  0,  0,  0,  "IC"   },
      {  0x45,  50,  - ,  2,  1,  1,  6,  4,  9,  0,  4,  4,  9,  1,  0,  0,  45,  00,  0,  0,  0,  0,  "BAL"  },
      {  0x46,  50,  - ,  2,  1,  1,  6,  4,  4,  0,  4,  4,  9,  1,  0,  0,  18,  00,  0,  8,  0,  0,  "BCT"  },
      {  0x47, 150,  - ,  2,  1,  1,  3,  1,  1,  0,  4,  4,  9,  1,  0,  0,  00,  00,  0,  0,  0,  0,  "BC"   },
      {  0x48,  50,  - ,  2,  1,  1,  1,  5,  3,  0,  1,  4,  9,  1,  0,  1,  15,  00,  0,  0,  0,  0,  "LH"   },
      {  0x49,  50,  - ,  2,  1,  1,  1,  5,  3,  4,  1,  4,  9,  1,  0,  1,  11,  00,  0,  0,  0,  1,  "CH"   },
      {  0x4A,  50,  - ,  2,  1,  1,  1,  5,  3,  4,  1,  4,  9,  1,  0,  1,  18,  00,  0,  0,  0,  1,  "AH"   },
      {  0x4C,  50,  - ,  2,  1,  1,  6,  6,  5,  0,  1,  4,  9,  0,  0,  1,  18,  00,  0, 32,  0,  0,  "MH"   },
      {  0x4D,  50,  - ,  2,  1,  1,  6,  4,  9,  0,  4,  4,  9,  1,  0,  0,  45,  00,  0,  0,  0,  0,  "BAS"  },
      {  0x4E,  10,  - ,  2,  1,  1,  6,  6,  5,  0,  2,  4,  9,  0,  1,  0,  11,  00,  0,  8,  0,  0,  "CVD"  }
      };
```

Figure 7

Example of the common E-unit/I-unit opcode table.

coverage, filling in holes (both known and unanticipated) that might have existed in the test-case generator.

At the core of the I-unit environment was the opcode table; a portion of the table is shown in **Figure 7**. The opcode table listed all opcodes that could be decoded and executed by the hardware, and provided a means for both driving the simulation and checking the correctness of the hardware.

There are two columns in the table that affected how the test case was constructed. One column was a relative weight, which affected the probability for a given opcode to be selected for decode. The other column controlled the probability that the selected opcode would have an I-unit-detected exception associated with it. The remaining columns contained information that the software monitor interpreted for verification of the hardware. The opcode table was shared by both the I-unit and the E-unit simulation environments. This allowed the team to detect interface discrepancies before the two units were built into a common model.

The software monitors in the I-unit verification environment were logically divided into two distinct pieces. The first piece ensured that the correct instruction was decoded. When the hardware issued an instruction fetch to a previously unused address, the opcode table was accessed a number of times to load an instruction stream at the fetched address. Once an instruction was correctly decoded, the monitors would then check that all data associated with the execution of the instruction was handled properly by the I-unit hardware. The monitors would do this by creating an instruction object for each decoded instruction. This object became the central reference point for all monitors in the environment. As the instruction advanced through the pipeline, the expected values would be extracted from the object and compared with the actual values generated by the hardware. In addition, some monitors would add expected data to the object that would be used in turn by monitors later in the pipeline.

Emphasis was placed on verifying functions that were not easily tested in a test-case-generation environment. Functions such as address-generation interlock (AGI), AGI bypass, serialization, address-mode changes, and program-event recording (PER) could be stressed more heavily in the unit simulation environment. In addition to having random rather than predetermined test cases, the team was also able to exercise greater control over the interfaces from the connecting units because the effect on those units did not have to be considered. This enabled the team to reach interesting corner conditions more easily.

The I-unit environment for this system also included the branch-prediction logic (BPL). On past machines, this was verified in a separate environment as well as at the element level. By incorporating the BPL in the I-unit environment, the team was able to reduce the work effort because separate BPL drivers were not needed. The stress on the BPL could be controlled by varying the weights of the branch instructions in the opcode table. In addition, by not separating the BPL from the rest of the I-unit, the team was able to find many complex bugs in the interaction between BPL, instruction fetching, and the decode logic.

E-unit verification

The methodology for verifying the E-unit was similar to that described in [1]. AVPGEN (an IBM tool originally developed for S/390 verification) was used to generate architectural verification programs (AVPs), and a random environment was built to drive the interfaces to the E-unit from the I-unit, BCE, and register unit (R-unit). Several enhancements were made to this process to verify the new features of this microprocessor E-unit. In particular, symbolic instruction graphs (SIGs) were created to stress the superscalar nature of the machine, including register dependencies, forwarding, and special dispatch grouping cases. An example of a simple superscalar SIG is shown in **Figure 8(a)**, and one with register dependencies is shown in **Figure 8(b)**.

The SIG in Figure 8(a) simply generated sequences of triplets of instructions that could be grouped together for dispatch. The E-unit had three execution pipes, the first of which was designated the `R-Path`. It could receive only certain branch instructions, which were listed in the `AnyRPathOp ()` macro. The other two pipes could receive superscalar instructions, which were listed in the `AnySuperScalarOp()` macro. Whether or not a group of three instructions was actually issued together was a function of the I-unit driver in the random environment.

The SIG in Figure 8(b) generated pairs of instructions that could be grouped together for dispatch, where the first instruction would forward operand data to the second. The first instruction modified a target register (designated as `R1`), which was used as a source by the second instruction (as either an `R1` or `R2` field). This would force the E-unit to do the forwarding.

A second enhancement to the E-unit verification methodology was the addition of coverage modeling. Several "cross-product" coverage models were created to measure the coverage of our tests. One model was the cross product of all possible groupings of instruction triplets. This model had over 250,000 coverage points (or *tasks*), and it took several months for us to hit all of them. To do so, we created a coverage feedback process by which we automatically adjusted the SIG macros to try to force cases that had not yet occurred. Other coverage models included all of the forwarding cases as well as all of the special grouping rule cases.

Other enhancements included new cycle-by-cycle monitors for several architectural features (condition code setting, branch resolution, and serialization), the mixing of special milliop instructions with z/Series instructions, and the creation of many special configuration files to bias the frequency with which the I-unit driver would optimally group instructions for dispatch, the dispatch rate, and the frequency with which other blocking conditions would occur (for example, a data miss from the L1 cache).

BCE unit verification

The BCE has evolved since the advent of the G4 system. The unified cache was split into a data cache (D-cache) and instruction cache (I-cache). The translation function became a unit of its own with two interfaces, one to the D-cache and one to the I-cache. To supply more data to the I-unit and E-unit, each cache has two pipes for requests based on address bit 55. The I-unit presents its requests without regard to address bit 55. The BCE handled routing the request to the correct pipe. A major challenge for verification was the introduction of out-of-sequence (OOS) fetching. The BCE must continue to honor and process requests, even when the older requests miss in the L1 cache. There are no controls in the BCE that would enable it to process the requests in order. To run the processor as fast as possible, an asynchronous interface was added between the processors and the SC.

The BCE was verified using the random methodology employed on G4 [1]. An important aspect of this methodology was that the interface drivers honored the interface protocol, but generally made no attempt to mimic the missing hardware. For example, in the real microprocessor, the translator can only be working on an I-cache request or a D-cache request. Since the logic contained in the BCE does not have this restriction, there are independent drivers for the I-cache-to-translator interface and for the D-cache-to-translator interface. This keeps both the I-cache and D-cache logic busy, which increases the probability of conflicts in the priority logic-to-the-L2 interface, thus giving the team better utilization of simulation cycles.

The drivers were developed with the ability to bias toward interesting behaviors. One area that was targeted was fetching from lines in transit from the L2. To improve performance, the BCE services requests to transit lines, returning data to the requesting unit when it arrives at the cache. By using information from the monitors, the drivers were able to generate requests to lines that were currently receiving data from the SC. To attack potential hangs, the monitors blocked the driver from doing stores until a randomly selected older fetch completed. Both of these approaches targeted areas where past unit simulation was weak and where OOS made bugs more likely. We were

```
/* Generate sequences of 3 instructions which may be dispatched together          */
#include <super.mac>       /* Defines AnyPathOp() and AnySuperScalarOp() macros    */

/* Create a macro to randomly generate a 3 instruction sequence                   */

macro Triple()

sig {
     AnyRPathOp() with NoBranch;  /* 'NoBranch' directive -> next sequential instr*/
     AnySuperScalarOp();
     AnySuperScalarOp();
    }

orcam

super: sig {   /* Randomly pick a number between 5 and 20, and generate that      */
               /* number of 'Triple' sequences.                                   */
                  sequence 5..20 of Triple(););
                  end
           }
start super;
```

(a)

```
/* Generate multiple sequences of  instructions, one of which will 'forward' its   */
/* result to the subsequent one.                                                   */
#include <super_fwd.mac> >  /* Defines AnyGRFwdS1Op() macro, etc...                */
macro Forward()  /* Create a new macro to generate forwarding 'Pairs'.             */
sig {
     AnyGRFwdS1Op() with x:R1;  /* Generate a GPR forwarding op; declare           */
                                  /* variable 'x', and associate it with tgt reg   */
     Oneof (
            AnyGRFwdT1Op() with R1=x;     /* Generate a GPR op which has the       */
                                          /* same source reg as the target reg     */
                                          /* of the prior op.                      */
            AnyGRFwdT1Op() with R2=x;
           );
    }
orcam
super_fwd: sig {
                /* Randomly pick a number between 10 and 30, and generate that     */
                /* number of forwarding 'Pairs' of instructions.                   */
                   sequence 5..20 of  Forward(););
                   end
               }
start super_fwd;
```

(b)

Figure 8

Symbolic instruction graphs: (a) Creating superscalar instruction streams. (b) Stressing register interlock.

successful in finding bugs that would have been more difficult to detect at higher levels of simulation.

Out-of-sequence fetching and asynchronous interfaces introduced design complexity that required increased internal monitoring to properly predict BCE responses. This monitoring tended to be more complex since internal unit signals were less rigidly defined; but this approach did allow for earlier detection of hardware problems, thus making the debugging process easier. Another benefit was the ability to aggressively change L2 data to stress multiprocessor (MP) testing. Knowing the details of the directory, the monitor was able to change L2 data earlier and more frequently, thus allowing the verification team to quickly and easily locate any MP data integrity problems.

The z/Architecture* allows for self-modifying code; when a store into the instruction stream occurs, this is referred to as *program store compare* (PSC). The split data and instruction caches complicate the logic for detecting PSC. The PSC escapes on the prior system illustrated a potential weakness in the random methodology. Even though the simulation monitor and the logic were independently designed, we ended up using the same algorithm, eliminating the independent check. A more pragmatic approach was used on the z990 machine. The checking was divided into three independent sections. The first part made sure that whenever a line associated with a valid instruction-fetch ID was lost from the I-cache, the I-cache informed the E-unit of an instruction-fetch ID invalidate. Losing the line was important because, to save bits, the PSC logic tracks a line by its location in the cache. The following actions caused the I-cache to lose a line: replacing a line with a new line, external XI from the SC, internal XI from the D-cache, or a purge cache command from the translator. The second part of the checking ensured that whenever a store occurred, the line was not valid in the I-cache. The monitor also ensured that stores to lines not held exclusively in the D-cache resulted in an internal XI to the I-cache. The third part made sure that an internal XI was reported whenever a request hit a valid instruction-fetch ID, and that the hit was reported to the E-unit when the store occurred. Unfortunately, the PSC verification code was not completed until after the first-pass silicon was released to manufacturing, and the test floor did find a PSC logic problem that would otherwise have been found in simulation.

This unit environment did a good job stressing the BCE design. The processor element environment was verifying the full processor, including the BCE, six months before unit simulation considered the D-cache functional. Until this point, nightly BCE unit regression runs exercised only the I-cache. Submitting short D-cache runs during the day located enough logic problems to keep the designers busy. This meant that there were D-cache problems pulled out at element level that normally would have been found at the unit level. One example of this was exceptions. By the time the processor element simulation was running cleanly enough that it was ready to start running AVPs with exception conditions, the unit simulation focus was still on getting the D-cache to consistently run clean short runs with basic fetches and stores without exception conditions.

In the end, there were four more test-floor escapes on this system than on the prior system (11 compared with 7). Out-of-sequence fetches and the memory book ring introduced an interesting class of hangs that were difficult to capture in a random environment. The hangs were related to cyclic behavior that developed in three- and four-book SC configurations with regard to XIs and data returns. Improvements were incorporated to go after these hangs, but more innovative work is needed in this area.

Processor element verification

As in [1], the processor model included the HDL design of the I-unit, E-unit, R-unit, and BCE, along with a software driver for the L2 and Licensed Internal Code for the more complex z/Series instructions. AVPGEN was used as the primary test generator for "mainline" function, and the same set of additional functions described in [1] was again verified. The following sections describe new methodology and testing unique to this microprocessor.

New architecture

Since this microprocessor supports trimodal addressing (24-, 31-, and 64-bit), it was important to stress all three, including the transitions from one to another. The z/Architecture includes instructions for switching to any of the addressing modes (SAM24, SAM31, SAM64). To verify these transitions, several special SIGs were written; an example is shown in **Figure 9**.

The load address (`LA`) in the SIG in Figure 9 is just to initialize a loop counter (which will have a value in the range of 4 to 8). The main part of the SIG generates a set-addressing code (SAM) instruction followed by a test-addressing mode (TAM) instruction, followed by some other instruction. That sequence is repeated several times to obtain a mix of different SAM instructions. Finally, there is a branch instruction with a target of the loop instruction. The constraints on the branch instruction force the branch to be taken the first time, and it uses the loop count (`R1` register) initialized by the `LA`.

Dual-core design

This design point included two processor cores on each processor chip. Even though the processor verification model consisted of a single processor, the team needed to take into account the dual-core nature of the chip. One way we did this was to create a dual-core driver that was a software behavioral program designed to create traffic on the bus from the other processor to the L2. Random commands and responses were sent at random times (with good parity) to try to stress the "real" core in our model. The dual-core driver was also used for recovery testing. The recovery algorithm for the machine requires communication between the two cores (assuming that both are functional). The dual-core driver was the vehicle that provided the handshaking for the nonexistent core. This allowed the team to completely verify dual-core recovery with a single-core model.

Operand buffer refetch

The operand buffer controls in the microprocessor design could "lose" data after it was returned from the L1 cache,

```
/* Generate a sequence of instructions which randomly sets the address mode,         */
/* followed by a test of the new mode, and some other random instruction.  Do this    */
/* in a loop, between 4 and 8 times.  Note: the TAM instruction sets the              */
/* Condition Code based on the address mode.                                          */

#include <anyop.mac> /* Defines AnyOp() Macro  */

sam: sig {
          LA with x:R1, B2=0, X2=0, D2=4..8;  /*  Load Loop Count  */
          loop: oneof(SAM24; SAM31; SAM64;);
              TAM;   /* Test Resulting Address Mode                        */
              AnyOp(loop);   /* Choose some random instruction             */
              oneof(SAM24; SAM31; SAM64;);
              TAM;
              AnyOp(loop);
              oneof(SAM24; SAM31; SAM64;);
              TAM;
              AnyOp(loop);
            /*Loop back                                                    */
          oneof (BCT loop; BCTR loop; BRCT loop;) with TakeBranch,  R1=x;
          end
         }
start sam;
```

Figure 9

Symbolic instruction graph: Transitioning between addressing modes.

for example, due to a XI from another processor in a symmetric multiprocessor (SMP) system. Because of the storage consistency requirements of the z/Architecture, the operand buffers had to refetch the data. This was a complex area of the design. To test it thoroughly, the team created a special refetch driver, which ran in conjunction with a refetch monitor to artificially invalidate the data in the operand buffers at random points in time, forcing a refetch to occur. This enabled the team to find complex MP bugs while having only a single core in the model.

Quiesce

Another complex area of the processor design was the quiescing of the system required by certain z/Series system instructions. The z/Architecture dictated that whenever certain instructions were run in an SMP system, the other processors had to "stop" while the processor executing the instruction completed the operation. An example of this was the *invalidate page table entry* instruction, which caused a change to a translation table entry. Other processors could not reference the table entry while it was being changed. There was an elaborate protocol for quiescing the processors, designed to ensure that the architecture was adhered to while at the same time optimizing overall system performance. To verify this protocol, a quiesce driver was created to emulate other processors in an SMP system, as well as the SC (the single point of control for the quiesce algorithm). The driver needed to handle both local and remote cases of the quiesce, as well as fast and slow versions of the algorithm (when an optimistic fast mode was rejected by the SC because of a collision, the algorithm dropped back to a slow mode).

Pervasive functions

To improve the verification of non-mainline areas, we incorporated much of it into our mainline regression. These areas included the following:

- *Disables:* The processor design included many latches that disabled a functional area of the design. For example, the BPL could be shut off entirely. These disables were included as potential workarounds for design bugs on the test floor and as degrade modes for recovery actions in a customer machine. There were more than 100 of these disable latches. We were able to randomly set them in our mainline regression and, in many cases, the team wrote special monitor programs to verify that we were disabling the intended functional area.
- *Address compare:* Address compare was a manual operator control that detected the fetching, storing, or

instruction fetching of a given address. The team wrote a program to randomly initialize the address compare settings (based on addresses in an AVP) and then wrote a monitor to ensure that the address compare was correctly detected.

- *Instruction step:* This was another manual operator control designed to cause an interrupt after each instruction executed. A monitor was created to make sure the interrupt occurred when in this mode and was run with the mainline regression AVPs.
- *Opcode compare:* Opcode compare was a control that caused one or more actions to occur when a particular instruction opcode was encountered: for example, to invoke Licensed Internal Code. Again, the team wrote a setup program and a special monitor to verify that the correct action was taken at the proper time, using AVPs in our mainline regression.
- *Forced serialization:* Certain events in the processor were defined to serialize architecturally. This meant that prefetched instructions were thrown away and refetched, and that instruction dispatching was reset and restarted. To test this feature, the team wrote a driver program that artificially forced serialization events to occur at random points and cleared the instruction buffers to ensure that the I-unit was refetching the instructions.
- *Random asynchronous interrupts:* Certain events in the system can cause an asynchronous interrupt (for example, a timer pop or an I/O interrupt). Again, the team wrote a driver program to test this feature that randomly forced asynchronous events at random points and ran this program with the mainline regression AVPs.

Array preloads
To make more effective use of simulation cycles, the team wrote programs to preload many arrays that would normally be loaded over time by the design. These included the L1 instruction and data caches, the TLB1 and TLB2 arrays, and the branch history table arrays. The loader programs were designed to load addresses from the AVP test cases. Using the loaders, the arrays could be loaded with all of the addresses from the AVP, a random subset of the addresses, or none of them. Preloading the arrays provided for more realistic simulation and better coverage. Since the arrays were not in the empty state prior to every run, the cache and buffer hit-and-miss behavior was different during the starting cycles of each run. This was especially important given the shortness of the runs in terms of real machine cycles. By preloading these arrays, our simulation runs started from a state that could otherwise take hundreds or thousands of cycles to reach.

Parameter files
Finally, to control the biasing of the many software drivers in the processor simulation environment, special parameter files (called *config files*) were created. One file biased the runs toward long delays from the L2 for fetch data, another file caused many XIs to be sent from the L2, and so on. To create interesting combinations of these config files in any particular test, the team created a type of config file that specified other config files in such a way that random cross products of these parameter files were chosen.

Coverage
In this project, coverage was approached from a more methodical point of view than in previous projects. Two different methods were used: cross-product coverage using Meteor, an internal tool which is an extension of Comet [6], and discrete-point coverage using a C++ class named `RndStats`.

At the beginning of the project, the team created a list of functions for which the completeness of testing was of particular concern. The list contained a combination of basic architectural features (for example, instructions in the architecture), newly implemented and complex functions (for example, the compression unit and translator), and areas that had been problems on past programs such as PER and start interpreted execution (SIE). The team also attempted to create these models at an architectural level in order to maximize the amount of reuse from one program to another.

Process of creating and designing
To implement coverage metrics on each functional area, the team went through several phases. First, we determined a detailed list of attributes of the function to be covered. Next, we created the database of coverage events for the domain and models in the Meteor tool. We then configured the environment and wrote programs and scripts to extract, format, and deliver coverage data to the Meteor server. After that, the team collected and measured coverage data, analyzed results, and redirected test-case generation to cover the holes we had located. We repeated this last step until the coverage goal for the domain (a group of attributes that describe a specific function) was met.

The following coverage domains were created:

- *PER coverage:* Program event recording is designed to assist in debugging programs. On the basis of an analysis of test-floor escapes from previous projects, the team felt it was important to create a coverage domain for PER. The team tracked several models (a *model* is a comprehensive list of coverage events related to a specific domain or logic function):

- Branch instructions compared with address mode, architecture mode, PER code, whether the branch target was within a specified storage range, and wrap condition.
- Store instructions (including store using real address instructions) compared with address mode, architecture mode, and valid PER code.
- Exceptions compared with all PER codes, architecture mode, and address mode.

- *Instruction coverage:* In the complex design of the z990 system, roughly 700 instructions are defined in the architecture. The team implemented a domain to verify that all instructions had been executed in all valid addressing, translation, and architectural modes. The team also verified that each instruction that could change the condition code had been simulated with all possible condition codes, and that we had tested all valid instruction and exception combinations.
- *Summary of basic architected features:* The purpose of this domain was to monitor our test-case generator. The team wanted to ensure that it could generate all possible hardware architectural modes (for example, all possible values for control registers, floating-point control register, and program status word).
- *Load real address (LRA) instruction:* The team tracked coverage on the multiple variations of the LRA instruction because, relative to other instructions, we handled address translation and exceptions differently. This detailed coverage model contained 105 *events* (tasks or logic states that should occur in simulation). Each event was made up of the combination of a mnemonic, a detailed exception, and the address mode.
- *Instrumentation:* Trace and instrumentation were debug assist functions that allowed the design engineers to collect data about the processor state and performance during execution. While the processor was active, data was written into several 64-bit arrays that could be periodically read and analyzed. A detailed verification of this function would have been very time-consuming, but we had to be sure that, at a minimum, there was real logic driving each bit. This coverage model ensured that each bit had been observed to have both a *0* and a *1* value.
- *Translator:* The new design of the translation logic, which was now a separate unit (described above in the processor verification section), made it a prime candidate for coverage metrics. The team created a domain to monitor our coverage for translator exceptions in combination with architecture mode and SIE mode. The team collected trace data for this coverage model from both the translator unit and the element simulation environments.
- *Compression call instruction:* This complex instruction had a large number of architected features, and the detailed coverage model contained about 2800 legal combinations. Some of the attributes included in this model were the condition code, source length, target length, and exceptions.
- *Superscalar grouping:* There were complex rules that defined which instructions could be executed simultaneously. The team created models that monitored three areas addressed by these rules. First, we ensured that all legal combinations of up to three instructions were observed. Next, we verified that certain groupings did not negatively affect performance. Finally, we ensured that certain grouping were given special treatment to enhance performance (for example, operand forwarding). The team spent considerable resources to reach 100% coverage on this model in both E-unit and processor element simulation.
- *Storage controller responses to processor commands:* This domain covered a fundamental function of the storage controller: to handle processor commands. The primary attributes in this domain were the processor commands and the SC response. This domain contained about 19,000 legal events.

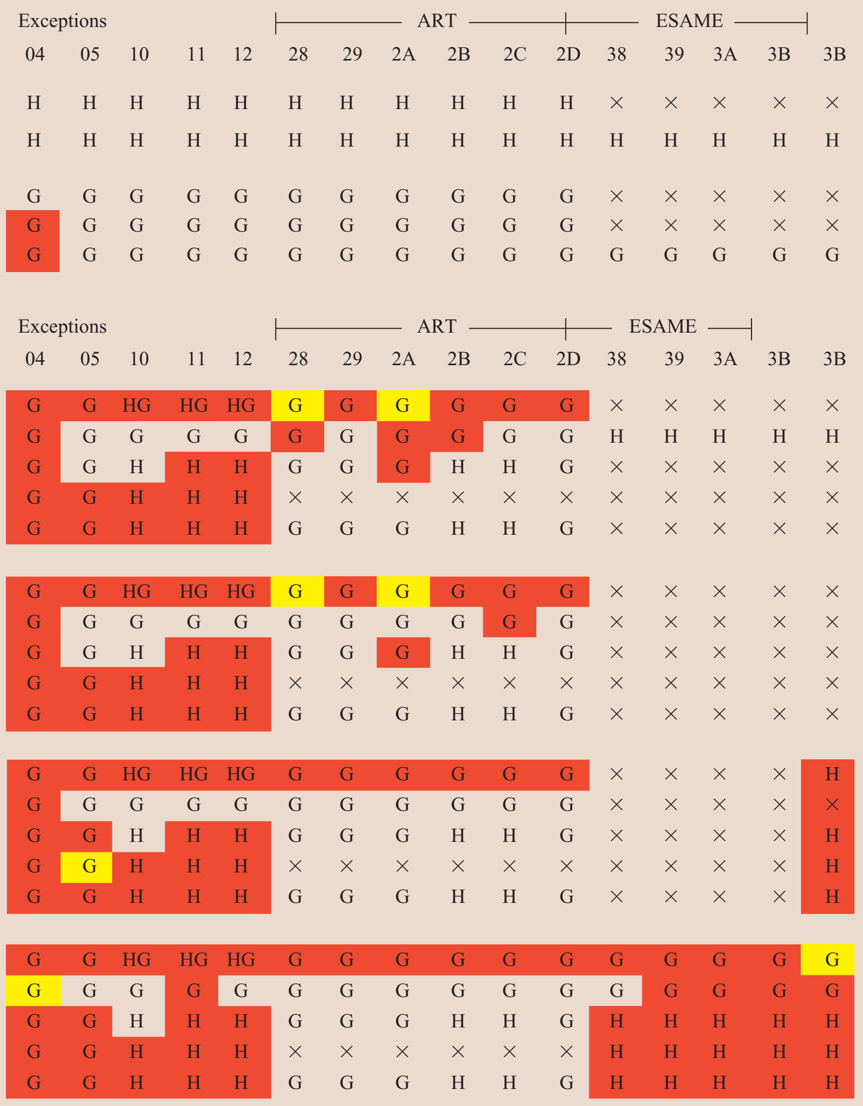
Findings and actions taken

Specific examples of some problems found included the following:

1. The LRA instruction was never tested with a specific exception.
2. Some instrumentation bits were never exercised.
3. The RLOCL instruction was never generated with a PER store exception.
4. An E-unit driver bug was preventing full coverage of operand forwarding.
5. Undirected test-case generation did not fully cover a model.
6. All valid combinations of floating-point rounding mode were not being generated.
7. The logic for detailed information during a response to some processor requests was missing.

To correct these problems, several actions were taken:

1. Extensive manual test-case redirection was done for several models.
2. Corrections and enhancements were made to AVPGEN.
3. Corrections were made to the software drivers.
4. Corrections were made to the software monitors (to correct holes caused by trace collection bugs).
5. An automated feedback mechanism was instituted in the E-unit simulation environment, described above.



Exceptions					ART						ESAME				
04	05	10	11	12	28	29	2A	2B	2C	2D	38	39	3A	3B	3B
H	H	H	H	H	H	H	H	H	H	H	×	×	×	×	×
H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
G	G	G	G	G	G	G	G	G	G	G	×	×	×	×	×
G	G	G	G	G	G	G	G	G	G	G	×	×	×	×	×
G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G

Exceptions					ART						ESAME				
04	05	10	11	12	28	29	2A	2B	2C	2D	38	39	3A	3B	3B
G	G	HG	HG	HG	G	G	G	G	G	G	×	×	×	×	×
G	G	G	G	G	G	G	G	G	G	G	H	H	H	H	H
G	G	H	H	H	G	G	G	H	H	G	×	×	×	×	×
G	G	H	H	H	×	×	×	×	×	×	×	×	×	×	×
G	G	H	H	H	G	G	G	H	H	G	×	×	×	×	×
G	G	HG	HG	HG	G	G	G	G	G	G	×	×	×	×	×
G	G	G	G	G	G	G	G	G	G	G	×	×	×	×	×
G	G	H	H	H	G	G	G	H	H	G	×	×	×	×	×
G	G	H	H	H	×	×	×	×	×	×	×	×	×	×	×
G	G	H	H	H	G	G	G	H	H	G	×	×	×	×	×
G	G	HG	HG	HG	G	G	G	G	G	G	×	×	×	×	H
G	G	G	G	G	G	G	G	G	G	G	×	×	×	×	×
G	G	H	H	H	G	G	G	H	H	G	×	×	×	×	H
G	G	H	H	H	×	×	×	×	×	×	×	×	×	×	H
G	G	H	H	H	G	G	G	H	H	G	×	×	×	×	H
G	G	HG	HG	HG	G	G	G	G	G	G	G	G	G	G	G
G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
G	G	H	H	H	G	G	G	H	H	G	H	H	H	H	H
G	G	H	H	H	×	×	×	×	×	×	H	H	H	H	H
G	G	H	H	H	G	G	G	H	H	G	H	H	H	H	H

Coverage progress for xlat-arch domain in CP element simulation

Red = Not covered 05/07/2003
Background = Covered without specific effort
Yellow = Holes left after one year of improvement

Figure 10

Translator element exception coverage.

Figure 10 shows the holes in the translator element coverage model. In this figure, G represents a guest exception, H represents a host exception, and × represents an illegal state. The red areas are holes that existed when coverage was first analyzed; yellow areas are holes remaining after all remedies had been implemented. Similar coverage information was also collected at the translator-unit level. There were a few holes left in both random unit and processor element simulation at the time of tape-out, but between the two environments, all combinations of exceptions were covered 100%.

Coverage infrastructure and tools

In the processor element environment, we decided to collect data for some of our coverage models by parsing the test-case files after generation and prior to simulation.

The advantage of collecting data by this method was that the collection mechanics were independent of design changes. The team took into consideration that we would be collecting coverage data on some failing test cases, but we decided that for our purposes, this was not important for two reasons. First, the sole purpose of coverage is to measure test completeness and efficiency; test correctness falls under functional verification. Second, the failing test cases were all screened, defects were opened on them, and fixes were eventually provided and verified. As a result, by the end of the project, all test cases ran successfully.

The team identified the test-case groups that best tested the function targeted by the coverage model and enabled data collection on these test groups. Each day, after the test cases were generated, the coverage collection engine was started. This engine consisted of a test-case parser class, a program that contains all of the subroutines to process the collected data for each of the coverage models, and a configuration table that controls the important parameters pertaining to this engine.

The parser extracted the architectural information from each test case—for example, instruction opcode, mnemonic, address, translation and architectural mode, absolute and virtual instruction addresses, absolute and virtual operand addresses, and so on—and passed this information to each of the coverage subroutines. According to the function under coverage test, relevant information was directed to a trace file. The trace files were sent to our coverage server and received in the coverage environment, where they were read by Meteor, the coverage measurement engine, and the coverage data was processed and recorded. The team used a configuration table to control and optimize the use of this environment and to enhance performance. Some of the features controlled this way were the enabling and disabling of the coverage data gathering and the number of trace files collected for each coverage domain.

The `RndCover` class of the random environment was created specifically to make it as easy as possible to collect coverage data during simulation. The general process used to enable data collection was to create a parameter file that described the model attributes. At run time, a parameter file determined how the data was formatted, where the data would be sent if the test case was successful, and, possibly, the source of the data in the simulation model. It also provided a method of controlling the amount of data per coverage model that was sent to the coverage server. At the end of the simulation, a postprocessor could be called to format the trace data, which was then sent to the Meteor server.

Two different mechanisms were used to determine the exact data to be collected: the *trigger* and the *C++ method.* The trigger mechanism worked by defining an event as a Boolean combination of simulation model facilities. For each cycle, when the triggering event was true, the value of each facility in the list was recorded in the trace file. The C++ method mechanism required the user to add code to a monitor or driver. The code calculated values and controlled the timing when the data would be written to the trace file.

Methodology discussion

Over the course of this project, the team discovered several things about this methodology that affected their efficiency and work flow:

- The majority of time spent working in the coverage arena is spent analyzing data, refining the coverage models, and redirecting test-case generation.
- It is good policy for the person who is doing coverage to be responsible for test-case generation as well.
- Keeping in mind that logic will change over the course of the project, the coverage engineer should build as much flexibility and extensibility into the coverage environment as possible.

Results and concluding remarks

As with any complex design point, the team had its share of design problems that escaped to the test floor. The test floor found 65 problems in the laboratory hardware; while this number was a bit higher than the goal, it represented less then 1.5% of the total number of problems found in the logic. All test-floor escapes were thoroughly analyzed by the verification team, and corrective actions were taken to enhance the verification environment to help prevent future escapes.

Two of the classes of problems that were prevalent on the test floor had failing scenarios that proved very difficult to recreate in simulation: MP hangs and multibook ring problems. The MP hangs were caused by problems in the new OOS logic located in the BCE unit. This was a focus area for both the design and verification teams, but bugs still escaped to the test floor. The multibook ring problems were discovered very late in the test-floor cycle. Again, this logic was a new design concept for this system. Once the verification team understood the underlying failing mechanism, we were able to enhance their checking programs to detect the failing mechanism without having to create the actual failure. This not only enabled us to find known test-floor escapes, but it also enabled us to find additional bugs in the multibook ring design.

The z990 team was satisfied with the verification results of this system. There were many new verification environments and methodologies created to fully verify and stress this design point. One example of a methodology change was the introduction of protocol verification. This marks the most extensive use of protocol verification thus far in the design of an IBM server system. The abstract protocol model for the z990 memory protocol was developed and refined over a period of two years, tracking the development of the protocol. In contrast, previous uses of protocol verification tended to be at a single stage in system development.

Modeling the protocol in an ongoing way allowed the z990 designers to test new features of the protocol before constructing the HDL implementation. The verification team determined that this approach improves the quality of the final design and saves valuable development time during the project.

Another important innovation was that the team, in its verification efforts, started to establish a connection between the high-level abstract model and the implementation. High-level properties derived from the abstract protocol model were implemented as checks on the HDL implementation in the random simulation environment. This allowed for more thorough checking of the implementation than would otherwise have been possible. For future projects, we intend to expand the connection between high-level and implementation-level verification.

This project was the first time the verification team seriously collected and analyzed coverage results. Given the large number of test cases that were run, there was a huge amount of coverage data being produced on a daily basis. We quickly realized that we would not be able to collect coverage data for every simulation run. The current coverage collection and processing tools were stretched beyond their limits. We had both reliability and scalability problems with the toolset. Run-time limits were developed to collect only a broad sample of the total coverage data. We have analyzed the bottlenecks in the current toolset and are correcting this for future projects so that more data can be collected and analyzed.

Designs are growing in complexity, and new verification tools, techniques, and technologies are constantly being invented by both vendor and in-house teams to address these complex design points. To increase the effectiveness of the verification team, the experience and innovation of its verification engineers will guide our choice of which tools should be used to yield a better design, which techniques should be applied to the problem set for the greatest benefit, and how new technology can improve the process while avoiding flashy trends that do not address a specific area of the process in a needed way. Knowledge gained from past experience and recognition of the directions in which new designs are changing increase the team's ability to prevent problems encountered in the past and to seek out new problems in the cutting-edge logic designs of the future.

Acknowledgments

The authors acknowledge all z990 design team members and other individuals throughout IBM for their contributions to the verification effort. We would particularly like to thank Sebastian Burckhardt, Kevin Calabrese, Mark Decker, Thomas Gilbert, Gary Hallock, Anne Huston, Patrick Duffy, Matthias Heizmann, Steven Licker, Anna Mozeshtam, Alia Shah, Klaus Keuerleber, Joerg Deutschle, Sascha Eckmann, and Wei-Yi Xiao.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Denali Software, Incorporated or The Open Group.

References

1. B. Wile, M. P. Mullen, C. Hanson, D. G. Bair, K. M. Lasko, P. J. Duffy, E. J. Kaminski, Jr., T. E. Gilbert, S. M. Licker, R. G. Sheldon, W. D. Wollyung, W. J. Lewis, and R. L. Adkins, "Functional Verification of the CMOS S/390 Parallel Enterprise Server G4 System," *IBM J. Res. & Dev.* **41,** No. 4/5, 549–566 (July/September 1997); see *http://www.research.ibm.com/journal/rd/414/mullen.pdf*.
2. J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. Chu, M. L. Behm, J. R. Baumgartner, R. D. Peterson, J. Abdulhafiz, W. E. Bucy, J. H. Klaus, D. J. Klema, T. N. Le, F. D. Lewis, P. E. Milling, L. A. McConville, B. S. Nelson, V. Paruthi, T. W. Pouarz, A. D. Romonosky, J. Stuecheli, K. D. Thompson, D. W. Victor, and B. Wile, "Functional Verification of the POWER4 Microprocessor and POWER4 Multiprocessor Systems," *IBM J. Res. & Dev.* **46,** No. 1, 53–76 (January 2002); see *http://www.research.ibm.com/journal/rd/461/ludden.pdf*.
3. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, MA, October 1992, pp. 522–525.
4. S. M. German, "Formal Design of Cache Memory Protocols in IBM," *Formal Methods Syst. Design* **22,** No. 2, 133–141 (March 2003); see *http://www.kluweronline.com/issn/0925-9856/contents/*.
5. Gary A. Van Huben, "The Role of Two-Cycle Simulation in the S/390 Verification Process," *IBM J. Res. & Dev.* **41,** No. 4/5, 593–599 (July/September 1997); see *http://www.research.ibm.com/journal/rd/414/vanhuben.pdf*.
6. R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User Defined Coverage—A Tool Supported Methodology for Design Verification," *Proceedings of the Design Automation Conference (DAC'98),* San Francisco, June 1998, pp. 158–163; see *http://www.sigda.org/Archives/ProceedingArchives/Dac/Dac98/papers/1998/dac98/pdffiles/09_2.pdf*.

Received September 22, 2003; accepted for publication December 1, 2003; Internet publication April 27, 2004

Dean G. Bair *IBM Systems and Technology Group, 522 South Road, Poughkeepsie, New York 12601 (dgbair@us.ibm.com).* Mr. Bair, a Senior Software Engineer, joined IBM in 1984. He received his B.S. degree in electrical engineering from the State University of New York at New Paltz in 1998. He has worked in the field of verification for 18 years and has verified multiple design points including I/O controllers, shared L2 cache designs, and microprocessors. Mr. Bair was the verification team leader for the z990 superscalar multibook microprocessor complex, as well as previous generations of zSeries machines. He has received numerous awards, including multiple IBM Outstanding Innovation Awards, and patents for his efforts in the field of verification. Mr. Bair is currently working on verifying future Systems and Technology Group design points.

Steven M. German *IBM Research Division, IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (german@watson.ibm.com).* Dr. German received his A.B. and Ph.D. degrees in applied mathematics from Harvard University in 1974 and 1981, respectively. He has more than 25 years of experience in many aspects of formal methods. In the 1970s, he developed the first automated system for proving the absence of common run-time errors in computer programs. After joining IBM in 1995, he originated a new field of verification algorithms for checking processor pipelines. Recently, he has focused on verification of multiprocessor memory protocols. Dr. German pioneered the Formal Design approach for developing hardware protocols, in which formal verification is integrated into the design process. He led the project to formally verify the memory protocol for the zSeries server and is currently verifying its successors.

William D. Wollyung *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (wollyung@us.ibm.com).* Mr. Wollyung is an Advisory Engineer. He joined IBM in 1974 and has been in verification since 1983. He received a B.S. degree in computer science from Lasalle University in 1998. He has worked on processor, storage controller, I/O, and memory verification, and is currently working on memory controller verification. Mr. Wollyung received a Gold Level Quality Award for his work in 9121 simulation, and IBM Outstanding Technical Achievement Awards for his work on 3090 S verification, S/390 G4 processor verification, and S/390 G5 storage subsystem development. He also received an IBM Outstanding Contribution Award for the G6 opera server development.

Edward J. Kaminski, Jr. *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (ekamin01@us.ibm.com).* Mr. Kaminski, a Senior Verification Engineer at the IBM Poughkeepsie facility, received his B.S. degree in electrical engineering from the Rensselaer Polytechnic Institute in 1987. He joined IBM at Poughkeepsie in 1987, working on symmetric multiprocessor (SMP) storage subsystem verification for the 3090 H2 design; he has continued in verification through subsequent generations of SMP designs to the current zSeries servers. Mr. Kaminski has received multiple awards for verification work, including an IBM Outstanding Technical Achievement Award.

James Schafer *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (schafer@us.ibm.com).* Mr. Schafer received a B.S. degree from Purdue University in 1982, joining IBM that same year. For his first eleven years with IBM, he worked on logic product diagnostic systems in East Fishkill, where he published several papers and received his first IBM Outstanding Technical Achievement Award. His functional verification career began in 1993 in Austin, working on various PowerPC support chips. Mr. Schafer started with the S/390 Server Group in 1996, and is currently a Senior Engineer working as team leader for system controller element verification. He has received multiple awards for his zSeries verification work, including an IBM Outstanding Innovation Award and an IBM Outstanding Technical Achievement Award.

Michael P. Mullen *IBM Systems and Technology Group, 522 South Road, Poughkeepsie, New York 12601 (mpmullen@us.ibm.com).* Mr. Mullen is currently a Senior Programmer. He received a B.S. degree in computer science from Union College in 1976, joining IBM that same year, and an M.S. degree in computer/information sciences from Syracuse University in 1981. Mr. Mullen has worked on the development of many mainframe systems; he is currently responsible for the hardware design verification of IBM z/Series processors. He has received several IBM Outstanding Technical Achievement Awards for his work in microcode development, the AVPGEN system, and hardware verification.

William J. Lewis *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (wjlewis@us.ibm.com).* Mr. Lewis joined IBM in 1982 and is currently a Senior Engineer. He received a B.A. degree in computer science from the State University of New York at Oswego. After starting out in the hardware performance area, he has worked in design verification since 1985. Mr. Lewis has received IBM Outstanding Technical Achievement Awards for G4 and G5 processor verification, and an IBM Outstanding Innovation Award for zSeries verification.

Rebecca Wisniewski *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (wisnski@us.ibm.com).* Ms. Wisniewski joined IBM in 1982 after receiving a B.S. degree in computer science from the College of Engineering, University of Illinois at Urbana–Champaign. She started in the hardware performance area, focusing on processor performance. In 1993, she began working in simulation on scalable POWERparallel adapters and switches. Since 1998 she has been doing buffer control element (BCE) unit simulation and is currently the zSeries processor verification leader. Ms. Wisniewski has received an IBM Outstanding Innovation Award for her work on z9000 verification.

Joerg Walter *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (jwalter@de.ibm.com).* Dr. Walter received his diploma in electrical engineering in 1986 and his Ph.D. in computer science in 1993, both from the University of Stuttgart, Germany. He joined IBM in Boeblingen in 1993,

working on memory card verification for the S/390 Parallel Enterprise Server G3. Dr. Walter is currently the verification team leader for the Boeblingen zSeries processor verification group.

Steven Mittermaier *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (mitt@us.ibm.com).* Mr. Mittermaier received an A.A.S. degree in electrical engineering from the University of Toledo in 1988, and a B.S. degree in electrical engineering in 1996 from the State University of New York at New Paltz. He joined IBM in 1988, working on photolithographic tools and support in semiconductor production. He later joined the CEC verification group in Poughkeepsie, working on processor simulation and becoming an expert on cross-product coverage. Mr. Mittermaier is currently working on processor verification and coverage for future Systems and Technology Group products.

Visda Vokhshoori *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (visda@us.ibm.com).* Ms. Vokhshoori received a B.S. degree in electrical engineering from the State University of New York at New Paltz in 1998, and an M.S. degree in electrical engineering from Columbia University in 2002. She joined the IBM z/OS component test group in 1998 as a consultant. In 2001 Ms. Vokhshoori joined the hardware verification group, where she is engaged in various verification projects with a concentration in functional coverage verification.

Robert J. Adkins *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (adkins@us.ibm.com).* Mr. Adkins is an Advisory Software Engineer. He has been in verification since 1985 and has worked on both processor and storage controller verification for multiple Server Group systems. He has received IBM Outstanding Technical Achievement Awards for his work on ES/9000 CP element simulation, S/390 G4 processor verification, and S/390 G5 storage subsystem development, and an IBM Outstanding Contribution Award for his work on S/390 G6 opera server development. Mr. Adkins currently works on the verification of future Systems and Technology Group microprocessors.

Michael Halas *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12603 (mohalas@us.ibm.com).* Mr. Halas received a B.S. degree in electrical/computer engineering from Rutgers University in 2001. He worked at Compaq Computer as a co-op student in 2000 on Tru64 UNIX clustering software and at IBM as an intern in 2001, joining the company as a full-time employee in 2002. He has worked on random verification of the BCE and a z990 I/O adapter chip. Mr. Halas is a verification engineer currently working on verification of the system controller for future Systems and Technology Group projects.

Thomas Ruane *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (ruane@us.ibm.com).* Mr. Ruane received a B.A. degree in English from the State University of New York at New Paltz in 1976 and an M.S. degree in computer science from Union College in 1980, joining IBM that same year. He is currently working on element simulation and tool support for follow-on z990 systems.

Ursel Hahn *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hahnurs@de.ibm.com).* Mrs. Hahn is currently an engineer in the IBM Server Group. She joined IBM in 1977 and is currently working on coverage, element simulation, and AVPGEN for follow-on z990 systems.