Functional verification of the CMOS S/390 Parallel Enterprise Server G4 system

by 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
R. J. Adkins

Verification of the S/390® Parallel Enterprise Server G4 processor and level 2 cache (L2) chips was performed using a different approach than previously. This paper describes the methods employed by our functional verification team to demonstrate that its logical system complied with the S/390 architecture while staying within the changing cost structure and time-to-market constraints. Verification proceeded at four basic levels defined by the breadth of logic being tested. The lowest level, designer macro verification, contained a single designer's hardware description language (in VHDL). Unit-level verification consisted of a logical portion of function that generally contained four or five designers' logic. The third level of verification was the chip level, in which the processor or L2 chips were individually tested. Finally, system-level verification was performed on symmetric multiprocessor (SMP) configurations that included bus-switching network (BSN) chips and I/O connection chips, designated as memory bus adaptors (MBAs), along with multiple copies of the processor and L2 chips.

Introduction

To verify the logical design of the S/390® Parallel Enterprise Server G4 (CMOS 4) processor and L2 chips before chip fabrication, our relatively small team of verifiers (hereafter designated simply as the verification team) defined the basic approaches that drove the verification effort. The initial focus was on the lowest levels of simulation, through which bugs could be removed as early as possible. This meant that our verification team assisted individual designers in creating simulation environments at designer macro levels, facilitating the removal of a large number of the bugs before traditional structured simulation (chip level) began. Throughout the effort described here, our team and the processor and L2 design teams were jointly responsible for the simulation of the design, which allowed for critical tuning of the environments that created test patterns and monitored for architectural and implementation compliance. Furthermore, this allowed for accelerated problem removal and bug-discovery-to-fix turnaround time. Finally, rather than having verification engineers assigned to work solely on particular verification levels, there was vertical movement of people across the four different levels. Not only did this enable our verification team to use their macro-level understanding of the design implementation, but it also allowed for environment portability as the model

0018-8646/97/$5.00 © 1997 IBM

scope increased across the levels. Furthermore, our team's shifting of effort from the lower levels to the higher levels corresponded to the maturity and functionality of the design.

Because the S/390 architecture is mature, a stable set of core tools was used for the architectural-level test generation. A strong architectural-level instruction stream test-case generator, AVPGEN, already existed [1]. Similarly, the random SMP methodologies used on prior S/390 storage controllers [2] were adapted and enhanced for the CMOS 4 storage hierarchy. Additionally, comprehensive escape analysis information from previous projects was used to direct verification, building upon knowledge gained from prior S/390 systems.

At the same time, advances in verification methodologies were used. The use of multiple simulation engines (hardware and software) for specific verification levels was coupled with a common application interface [3], SimAPI, which allowed for reuse of code across the platforms. Random and directed random drivers targeted at the design implementation were developed and utilized from the start of the program. TIMEDIAG/GENRAND, a tool set that uses timing diagrams to drive general or specific test patterns, was developed for the designer macro level [4]. New modeling techniques allowed functional verification to expand its boundaries to include full scan-ring, clock, and built-in self-test (BIST) testing in cycle simulation [5]. Performance improvements in proprietary cycle-simulation tools continued to increase the magnitude of simulation cycles available per unit of time.

With these strategies in place, the goals for the verification effort were set. While it might have been noble to strive for zero defects in the design, it was not a productive business goal given the current simulation methodologies. The time it would take to remove the last handful of bugs is considered to be better spent by learning from the fabricated chips. Therefore, the verification goals were set with our primary objective in mind: time to market. Verification, working closely with design, delivered a solid functional design that would allow the final problems to be removed in hardware so that learning about circuit design, timing, and logic correctness would proceed in parallel. Success would be indicated by the functionality of the first hardware release and the ability to work around the complex problems that remained in the design. In light of these goals, firm release criteria that supported functional progress were defined and enforced.

Verification methods

- *Simulation engines*

Verification of the logical functions was performed with both event and cycle simulators, as well as hardware acceleration engines. The choice of simulation platform for a given test depended mostly upon model size, model build time, and performance needs, although certain tests required features that were provided only in event simulation.

Since most test-case coding was done using the SimAPI user interface, detailed benchmarks on simulator performance were conducted that allowed both test-case interface tuning and analysis of platform efficiency for a given test suite or level. In general, event simulation was used only at the designer macro level, where model build time was fast and simulator performance was acceptable for small models. A commercially available VHDL event simulator was used. Unit- and chip-level verification were performed with proprietary cycle simulators, TEXSIM and ZFS. System-level verification used ZFS cycle simulation and three Engineering Verification Engine (EVE 1.5) hardware accelerators.

Cycle simulation generally provided a performance improvement of more than 100× over event simulation. The choice of cycle simulators, TEXSIM or ZFS, depended mostly upon the latch-switching factor.[1] Because TEXSIM and ZFS use different algorithms to perform cycle simulation, the latch-switching factor was the indicator of which simulator performed better for a given test and model. In general, such performance is inversely proportional to model size, because a small model can be driven harder with implementation testing than a larger model that has architectural restrictions. As a result, larger models such as the processor chip used ZFS for cycle simulation, while smaller models used TEXSIM.

Hardware acceleration was used for the largest of models, where performance can be achieved only with specialized systems. EVE 1.5 hardware accelerators [6] were used to run extended test streams on mature system models. These tests achieved speeds up to 20 times faster than the cycle simulators. The relative performance of simulators used in the verification process is shown in **Table 1**.

The event simulator, TEXSIM, and ZFS were all used on a pool of RISC System/6000* workstations. ZFS was also run on S/390 server engines. Overnight batch capacities on an average night after the design was stable allowed for a cumulative 100 million cycles of processor chip testing, 15 million cycles of L2 chip testing, and 150 million cycles of unit testing. EVEs gave an additional capacity of 65 million cycles a day.

- *Test-case types*

In the past, test cases were software code that stimulated the logic model and were handwritten by verification experts. A test case could be viewed before simulation to see exactly what stimulus would be applied to the model.

[1] The average percentage of latches changing their values per cycle in the test case.

Test-case libraries were maintained for regression purposes so that interesting patterns could be rerun to guard against breakage in the logic. In S/390, test cases have evolved in two directions, test-case generators and test-case drivers, as shown in **Figure 1**.

- *Test-case generators*

Test-case generators are used to create numerous hard-coded test cases. These generators are sophisticated software engines that can be focused on very specific scenarios or broadened to cover a wide range of logic. Thousands of test cases can be created in the time it used to take a verification engineer to write just one test case. And because the focus can be changed from narrow to wide, the generators can be used with a shotgun or a sniper approach to uncovering bugs.

The role of verifier has changed along with the test-case generators. The verifier used to be anchored at the architectural level of the design, having to write many interesting test cases by hand. Because writing test cases this way was time-consuming, it was difficult to touch the microarchitecture implementation. The efficient use of test-case generators creates two roles for the verifier. The first role entails maintaining the generator itself, including adding new features and updating the prediction software within the generator. The second role is that of test-case writer, in which the verifier studies the microarchitecture and creates templates for the generator. These templates create hundreds of different test cases that stress the implementation of the logic, creating conditions such as "buffer full," "pipe stall," or "unavailable resource."

- *Test-case drivers*

Test-case drivers do not create test cases that are viewable prior to simulation. Instead, they consist of software that drives the model's interfaces using the parameter settings for the particular run. Test-case drivers use pseudorandom coding techniques to choose from the parameter lists. At the heart of the test-case driver is the prediction-checking software that monitors the interfaces and flags error conditions. The checking is done in real time so that race conditions need not be predicted up front. In this mode, the inputs to simulation are merely a seed and a parameter file.

Test-case drivers run for a predetermined number of cycles. The test case ends either when an error condition is detected or when the predetermined number of cycles have been successfully run and the model is quiesced. In either case, while a readable test case is not available before the run, a full history of the test case is logged by the driver. All interesting actions that occurred during the run can be viewed in this history, with even more information than that within a hard-coded test case.

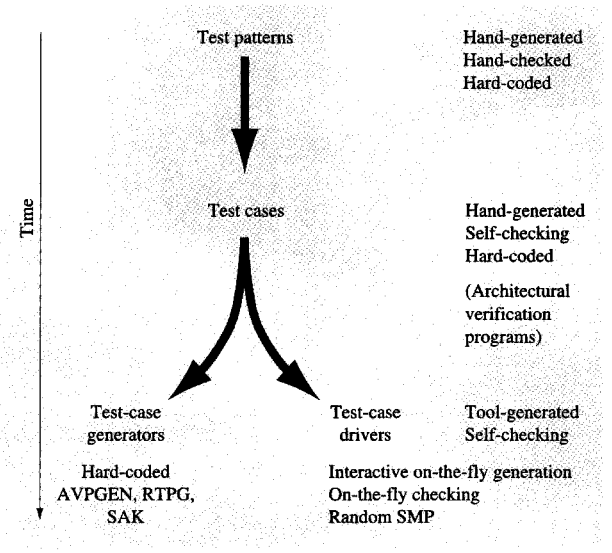

Figure 1

Evolution of test-case methodology.

Table 1 Relative performance of simulators.

Level	*Relative model size*	*Full model build time*	*Performance* (cycles/s)
Designer macro (event simulation)	1	2 min	40
Unit (TEXSIM)	30	1 hr	240
L2 chip (TEXSIM)	35	1 hr	220
Processor chip (ZFS)	120	4 hr	120
System (EVE)	1180	4 hr (postprocessor chip build)	380

There are three main advantages to test-case driver verification. First, the drivers are not architecturally restricted, allowing the behaviorals to author sequences that target the implementation of the hardware. As a result, the test cases can be far more stressful than conventional hard-coded tests. The second advantage is that the results of the complex internal conditions created during the test case need not be predicted before run time. This allows the monitors to make decisions on result validity after the race conditions have been resolved in the hardware. A main advantage of this is that there is no test-case library maintenance problem when internal timings change. The third advantage is that the tests can run as long as desired, with maximum stress throughout all

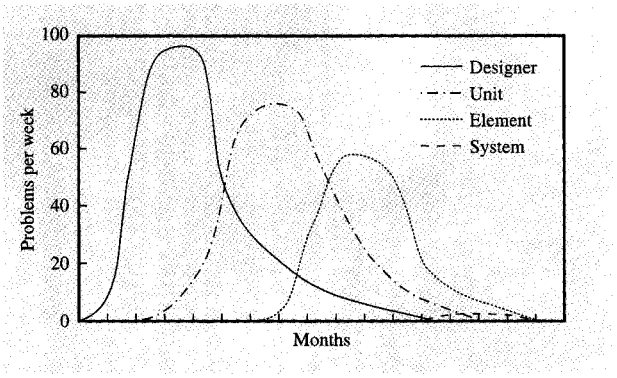

Figure 2

Problem rate for each level of verification.

of the cycles. This allows many "hard-to-create" conditions to occur during the test case. Cases of cache LRU castouts, timeouts, hang conditions, and lockouts are all more likely to occur in longer-running simulations.

Verification engineers using test-case drivers must perform the same types of work as those who support test-case generators. Software maintenance is required when interface specifications change, new commands are added, or additional result checking is needed. The behaviorals that drive the model must have the intelligence to understand the internal implementation as well as the interface protocol. By updating the behaviorals and adjusting the parameter files, the verifier creates new and interesting conditions within the logic.

- *Verification levels*

Verification experts were involved with the design before VHDL was even available for simulation. Each unit had a verifier working with the designers of that unit (approximately one verification engineer to every three designers). The verification expert served as a mentor for the designers in that unit as, together, designer macro simulation was performed. At the same time, the verifier was creating a unit-level environment that would be ready for unit-level testing as soon as the macros within the unit passed the readiness criteria. As the unit environment began running, the verification engineer began to focus on the chip-level environment. Here, code sections such as cache loaders and chip interface behaviorals that were created for the designer macro- and unit-level tests were reused for the chip-level environment. Thus, time and effort were saved through the planned reuse of code. In this manner, insights about the design learned at the macro level were carried throughout all of the higher levels of verification.

System-level testing required earlier planning than the paradigm used at the lower three levels, where verifiers moved with the design as it matured. Aspects such as multiple design language simulation and the host-to-EVE interface required extended tool development time that would not fit into the above "just-in-time" structure of personnel movement across the levels. Therefore, system-level preparation work began in parallel with the designer macro level.

The verification team comprised just twelve people. For the most part, the team designed and authored the environments needed for unit-, chip-, and system-level testing, while problem debug was shared by both designers and verifiers. As a result of this teamwork, fix turnaround time became far shorter than that of previous machines. At the point when unit testing was subsiding and chip testing was starting, it was not uncommon to have five or more problems identified, fixed, and verified each morning after the previous night's batch runs. Historically, prior projects had error rates that topped out at 20 bugs per week in the processor. The peak bug rate on this program was 60 bugs per week for the processor-level verification. Additional problems were simultaneously screened out (especially breakage) by the lower-level tests that were still in place to provide regression vehicles for VHDL changes due to timing or logic fixes. The result of this was that the models became fully functional relatively quickly. The problem rates for each of the levels of verification are shown in **Figure 2**.

Designer macro verification

Implementation verification on the smallest portions of the design has proven to be an effective alternative to the slower and often less stressful chip-level architectural testing. Today's leading-edge verification methodologies, such as formal verification, are geared toward implementation verification on smaller models. However, since production-quality formal verification tools were not available when designer macro-level testing was ongoing, the design/verification team had three choices for implementation testing:

- VHDL test benches or hard-coded SimAPI test cases.
- C/C++ program that drives patterns and checks results.
- TIMEDIAG/GENRAND.

The logic under test dictates the method chosen for designer macro-level testing. Complex control logic, for example, requires a more rigorous environment than simple dataflow logic.

The instruction unit's operand compare logic was thoroughly tested using a straightforward VHDL testbench. This testing consisted of the use of hard-coded patterns that cycled through the interesting opcode

compare logic cases and checked for correct results. Although limited in scope, this type of testing was sufficient for certain logic macros.

More complex control logic required the use of more sophisticated drivers and checkers. For these macros, C/C++ code was used to generate the interesting scenarios required to verify the logic. Often these programs used some random-pattern-generation techniques. The bus interface logic in the L2 chip used such an approach. In this case, 1500 lines of C code were written to drive the interfaces and check the results. Routines such as "Do_L2_Fetch" and "Do_LRU_Castout" stimulated the bus interface control logic with requests for actions. Other code routines, such as "LoadCastOut" and "Empty_CastOut," performed as behavioral logic that responded to the direction of the bus interface logic. These routines replaced other internal L2 macros that act as slaves to the bus interface logic (that is, they do not independently create their own stimuli, merely respond to requests made upon them). Control routines such as "Select_Op" were used to arbitrate among the requesters to the bus interface logic. Random-pattern generators created unique data to shuffle through the data paths. Finally, check routines such as "Check_L3_L2" ensured that the bus interface logic acted as expected.

As in the case of the C programs written for specific macros, TIMEDIAG and GENRAND were used to create high-stress environments that fully test the internals of the macro. TIMEDIAG and GENRAND allowed designers to utilize a pseudorandom methodology after creating generic interface protocol timing diagrams. TIMEDIAG, the timing diagram editor, allows the designer to make one or more timing diagrams that are used by GENRAND, the simulation driver, to create complex scenarios. Each timing diagram describes one or more actions on an interface, including expected results. Timing diagrams can be simple or complex, with looping conditions, random values, complex expressions, and particular start-up conditions. GENRAND uses this information to learn the interface protocols and then drive the timing diagrams pseudorandomly within the limits of the interface rules.

There were several advantages to creating the simulation environments just described. The greatest was that the chosen methodology was specific to the implementation, creating more stress on the logic than any other level achieved (including the actual hardware test environments). The most complex of conditions were created with relative ease at the macro level. Another advantage of these environments was the ease of regression. While most of these C environments took a month to create, the time was more than returned in ease of regression and speed of bug removal. Throughout the project, whenever timing, physical, or logic changes were made, these environments quickly verified the changed

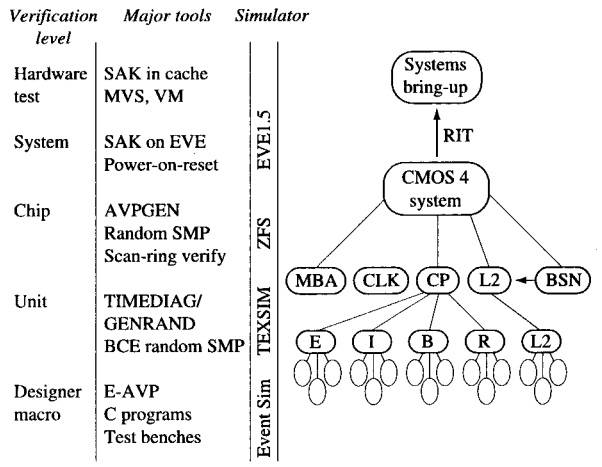

Figure 3

Verification levels. Verification proceeded from the designer macro level through the system level; at each level, environments and platforms were used that were best suited for that level.

VHDL to ensure correctness. For those macros where these methodologies were feasible, all of the higher levels of verification (unit, chip, and system) primarily became tests of the communications among macro interfaces.

Designer macro-level verification was performed primarily on the event simulator. Small units of logic fit well into the event simulator process because smaller models can be created quickly, and, since the model is tiny, the speed of the simulator is tolerable. Furthermore, the on-screen source-code debugger and enhanced graphic capabilities were welcome features for macro-level debug and logic analysis. The verification tools used at the designer macro level as well as the unit, chip, and system levels are shown in **Figure 3**.

Unit verification

Unit-level verification varied according to the function being tested. Two areas that required sophisticated methodologies at the unit level were the execution unit (E-unit) and the buffer control element (BCE). Investment of time and resource into these environments was high. The E-unit environment used the test-case generator approach because the generator already existed and was to be used at the chip level. The BCE unit environment used the test-case driver approach in order to bypass architectural (instruction stream) restrictions and attack the BCE implementation. Both methodologies were successful; the chip-level testing produced a low volume of problems in both of these units. The methodologies are explained in this section.

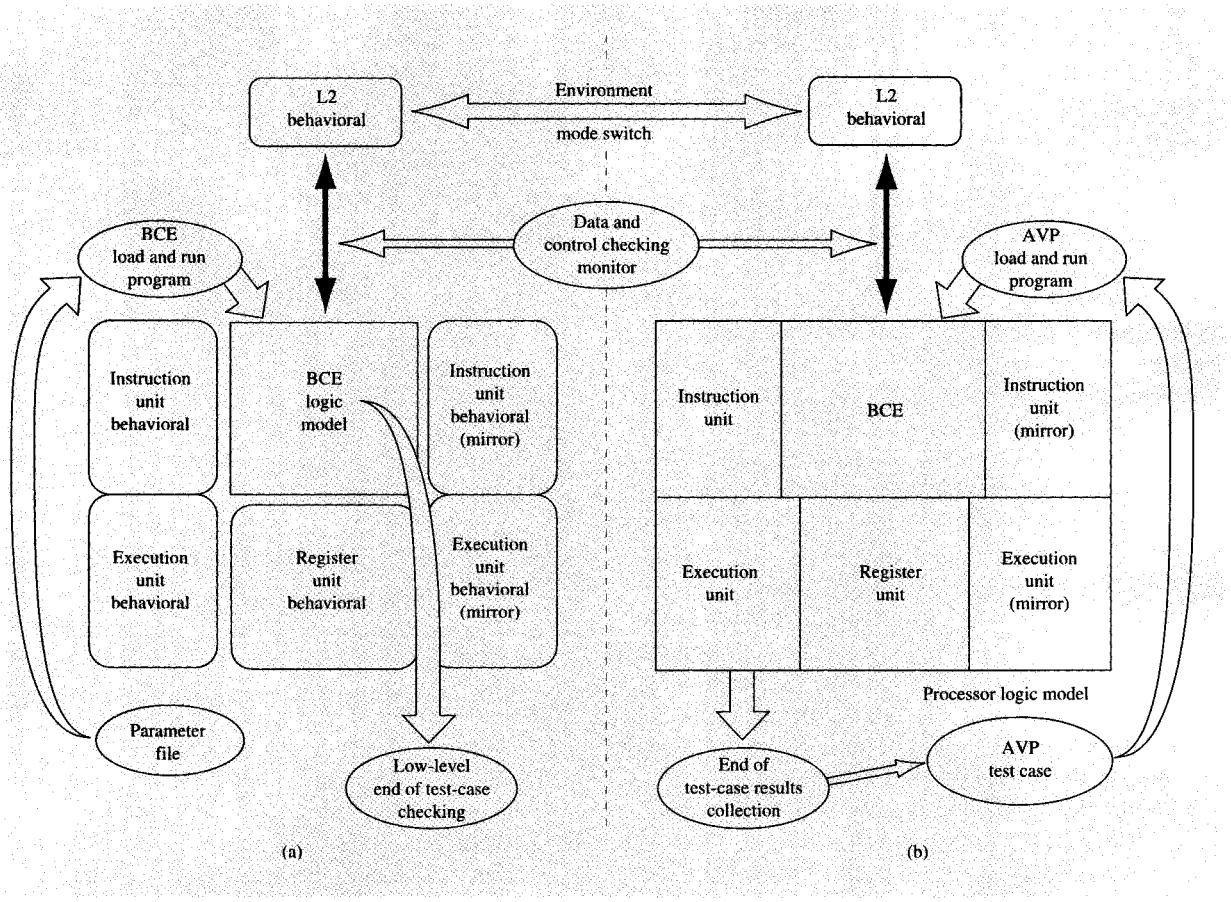

Figure 4

(a) BCE unit and (b) processor environments. The L2 behavioral code was used for both the BCE unit-level test-case environment and the processor chip-level verification environment.

E-AVP To efficiently verify the E-unit, an environment was created to use architectural verification programs (AVP) with a stand-alone E-unit model running on the event simulator. This environment, designated E-AVP, consisted of a set of programs to interpret the instruction stream in an AVP, and to "drive" the instruction unit (I-unit) and BCE interfaces to the E-unit. The programs also monitored the E-unit/BCE interface to record any data transactions that were done. Actual results for registers and storage were checked against the expected results in the AVP.

By running real instruction-stream tests at the E-unit level, complex E-unit problems were quickly discovered and fixed. Also, the same test-generation tool (AVPGEN) was used at both the unit and chip levels. The logic designers were able to use both AVPGEN and E-AVP themselves, tailoring the AVPs for the instruction streams in which they were interested. In order to drive the E-unit correctly with the instruction stream, the E-AVP programs accurately modeled the controls in the I-unit for decoding instructions and fetching operands, and the fetch and store controls in the BCE. By having programs control these interfaces, the user controlled the random biasing of certain key parameters such as the length of time between decodes, the delay on an operand fetch, and the amount of time that the BCE was busy for a data transaction.

BCE (L1) random Historically, the BCE has been the most difficult unit in the system to verify. On past microarchitectures, the BCE had the largest number of simulation escapes into the hardware bring-up. The problems reflect the high complexity intrinsic to any S/390 L1 cache and control, which must handle multiprocessing requirements for the processor, address translation, and multiple cache requests from the instruction unit. Therefore, thorough BCE verification was performed at

the unit level on the CMOS 4 S/390 program. The test-case driver methodology was used to accomplish this task.

The BCE consists of a store-through L1 cache, dynamic address translator (DAT), access register translator (ART), translation lookaside buffer (TLB), ART lookaside buffer (ALB), cross-invalidate (XI) stack, read-only system (ROS) array, and store buffer. The BCE has interfaces with the I-unit, E-unit, register unit (R-unit), and L2, as shown in **Figure 4**. In the simulation model, everything except the BCE was modeled as C++ behaviorals. These behaviorals were responsible for driving requests into the BCE and responding to BCE requests. The behaviors were programmed to obey interface protocol specifications and user parameter files. All behaviors shared a common address space that was generated at the beginning of each simulation run. The addresses that were generated for each run caused different levels of cache contention depending on the parameter file, which dictated the range of the number of addresses and the level of cache contention for any one run. The address-space generation code was used in the BCE, L2, and processor simulation environments.

The S/390 architecture has many different address-translation modes. In order to test both address translations and cache contention, multiple virtual addresses were mapped to the same absolute address. The architecture also contains a common segment (CS) bit, which was verified by mapping one virtual address to multiple absolute addresses. A virtual address mapped down to one of two absolute addresses, depending on the state of the CS bit in the TLB. The L2 behavior was responsible for responding to BCE requests and sending random XIs to the BCE. This allowed simulation of multiprocessor contention with only one BCE in the configuration. When the BCE no longer required data that it had used (released ownership of a line), the L2 behavior updated the address space with new data. This emulated the many cases where a second processor stored into a line in which the BCE currently holds.

The I-unit, E-unit, and R-unit behaviorals were responsible for initiating new requests to the BCE. Multiple parameter files were used as input to these behaviorals to stress different parts of the BCE. The section on L2 verification presents a detailed description of how the behaviorals used the parameter files.

In addition to the behaviorals, use was also made of automatic checking routines which were responsible for ensuring proper operation of the BCE. The checking routines dynamically updated expected results on the basis of events occurring in the simulation model. The checking routines verified such elements as cache coherence protocols, BCE-generated responses, translations, and interface protocols.

```
/*
  Store 4 bytes of an RX op over some other RX op.
  Do the store a little away from where we are.
*/
psc3 : sig
  {
   n1: L with NoException, x:R1, Mem2Start = psc.n3;
       ST with NoException, R1 = x, Mem2Start = psc3.n2;
       sequence (1..2 of AnyNonBranchOp() with NoException;);
   n2: AnyRXOp() with NoException;          /* 4 bytes of (OLD) op */
       sequence (1..2 of AnyNonBranchOp() with NoException;);
       end
   n3: AnyRXOp() with NoException;          /* 4 bytes of (NEW) op */
  }
```

Figure 5

Storing into the instruction stream.

Chip verification

Processor verification The processor model consisted of VHDL design for the I-unit, E-unit, R-unit, and BCE, along with the L2 behavioral (described in the unit-level verification section and shown in Figure 4) emulating the memory hierarchy. The model also included the licensed internal millicode (LIC), which was used to implement some of the more complicated S/390 instructions. The chip-level model was configured as a uniprocessor but was controlled at times, through the L2 behavioral, as if it were an SMP. This enabled random cross-invalidates (XIs) to the BCE from the L2 behavioral that allowed data- and instruction-stream contention typical of multiple-processor environments. The main verification strategy used at the chip level was random-biased testing, a methodology that has proven to be effective for verifying processor designs [7]. AVPGEN, a random-biased test-case generator, was used heavily in the verification of the CMOS 4 S/390 processor [1]. Many symbolic instruction graphs (SIGs) were created to stress specific types of S/390 instruction operations.

AVPGEN test cases covered the majority of the hardware function. These test cases were augmented in two ways. The first was with fixed AVPs, including both legacy AVPs and new AVPs targeted for specific functions. The second method used to increase the scope of verification was to alter the environment in which the random AVPs were run. This was accomplished with parameters that were randomly selected when the test cases were executed. Examples of the functions that were tested concurrently with the AVPs are cross-invalidates, quiesce, forced serialization, trace/instrumentation, error injection, and degraded/disabled modes.

```
/*
   Test Address Generation Interlock among several instructions.
   Interlock on both base and index registers.

*/
agi : sig
{
   n1: L with NoException,firstR1:R1;
       oneof( LA with secondR1:R1,X2=firstR1;
              LA with secondR1:R1,B2=firstR1;
            );
       L with NoException,thirdR1:R1,B2=secondR1;
       oneof( AnyRXOp() with X2=thirdR1;
              AnyRXOp() with B2=thirdR1;
            );
       AnyNonBranchOp();
}
```

Figure 6

Stressing register interlocks.

```
/*
  Test serialization by using SACF to switch Address Space bits.
  followed by any serializing op, followed by other ops.
*/
serial : sig
  {
    n1: sequence (1..2 of SACF; ) with NoException;
        AnySerializerOp() with NoException;
        AnyOp(t1) with NoException;
        sequence ( 1..2 of SAFC; ) with NoException;
        AnySerializerOp() with NoException;
        AnyOp(t1);
  }
```

Figure 7

Control instructions followed by dependent instructions.

The majority of the simulation effort went into verifying the mainline function of the processor (the term *mainline* refers to normal S/390 instruction execution). Nonmainline functions included resets and recovery.

The mainline verification consisted of the following strategy:

- *AVPGEN testing*
  Approximately 60 000 AVPGEN test cases were run nightly. The AVPGEN test cases were generated daily from a collection of over 60 SIGs. Some examples of these SIGs were the following:
  - Complex branch sequences.
  - Storing into the instruction stream (see **Figure 5**).
  - Stressing register interlocks (see **Figure 6**).
  - Control instructions which change the processor state, followed by instructions dependent on the new state (see **Figure 7**).
- *Fixed AVPs*
  The legacy AVPs were a subset of the test-cases used on previous S/390 processors. In addition to the legacy AVPs, a limited amount of new test-case development was done. This development occurred when AVPGEN and legacy AVPs did not cover a particular instruction, or when a legacy AVP required overhaul due to machine implementation. In all, approximately 25 000 fixed AVPs were regressed weekly.
- *Timing facilities*
  Use was made of a C behavioral program that was written to emulate the timing signals from the MBA chip (e.g., time of day update and synchronization). This program was used in conjunction with AVPGEN and fixed AVPs to verify the S/390 timing facility instructions and interrupts.
- *Interrupts*
  Programs were written to inject external and I/O interrupts. AVPs were modified to control the injection (type and event), and were run with these programs. These AVPs also checked that interrupt masking worked correctly.

Functions tested outside the mainline environment included the following:

- *I390 mode verification*
  I390 is special code that handles the service processor interface to the system. This testing used a fixed set of AVPs that focused on entry and exit from I390 mode, storage access, I390 special instructions, and interrupts.
- *Recovery verification*
  Error detection was tested by randomly injecting errors into the design while running a mainline AVP and checking to see whether the error was detected. Error recovery was verified by continuing the error-detection test case and checking to see that the system was successful in recovering from the injected error. These test cases were intricate in that the timing on certain injections was key to the recoverability of the logic. Injections that caused data integrity to be compromised were flagged to ensure that the logic halted the erroneous data propagation.
- *Scan-ring testing*
  Chip-level scan-ring verification was performed using the process described later in the section on scan-ring verification.

- *Trace and instrumentation testing*
  The trace and instrumentation functions were verified via a monitor that ran concurrently with AVPs. The controls were set up randomly at the beginning of the run. The monitor was a C program that was called each simulation cycle. The model facilities were examined and evaluated, and the expected data were put into program copies of the trace and instrumentation facilities. Each time the array filled up, as well as at the end of the test, the program data were compared to the actual data.

Each morning a regression report was generated and the failures were analyzed. A team screening approach was adopted, with the "screen team" consisting of both verification engineers and key logic designers. The designers were assigned to look at problems that related to their logic area. The "screen team" strategy worked extremely well and was one of the major reasons that problem turnaround time consistently averaged less than one day. As a result, a large number of bugs were removed from the design in a very short period of time.

L2 verification The L2 chip contains a second-level cache and associated dataflow and control logic. It interfaces with multiple processor chips and with the bus-switching network (BSN) chips, which provide a gateway to the L3 storage arrays. The L2 chip services data requests from the processor and BSN chips and maintains cache coherency within the multiprocessor system.

L2 chip-level verification was accomplished by applying the random SMP methodology used on prior S/390 storage controllers as a base [2], and by using experiences on past machines to enhance the scope and the efficiency of the simulation. While the goal of L2 chip-level verification was to ensure the functionality of the L2 chip itself, the simulation methodology on this CMOS 4 S/390 processor was expanded to allow the additional incorporation of several BSN chips, producing an L2–BSN multichip simulation model. Because the L2 and the BSN chips were designed at two different sites, the chance of interface protocol misunderstanding was greater. The L2–BSN multichip simulation model provided the ability to verify the interface between the two chips prior to system verification. In addition, since the random SMP methodology provided maximum stress of the functions within the L2 and BSN chips, it was an excellent way to achieve additional simulation on BSN chip functions.

L2 and L2–BSN chip-level simulation was performed using the TEXSIM cycle simulator. The L2–BSN chip-level model was built using a mixed design language process, since the L2 chip was designed in VHDL and the BSN chip design was not (see the subsection on mixed-language simulation). The core of the random SMP methodology is in the test-case drivers and automated checking programs, which are commonly called the "simulation environment." These programs were developed using C++ object-oriented techniques, enabling easy reuse of code across different levels of simulation. The following C++ objects were developed and reused across more than one level of simulation:

- *Address space*
  The address space object maintained a full set of addresses to be used in each test case, the latest copy of data for each of the addresses, and other relevant information pertaining to the addresses. This object was incorporated into the BCE unit-level, L2 chip-level, and CP chip-level simulation environments. It was referenced by the test-case drivers and automated checking programs whenever address-specific information was required.
- *Parameter list interface*
  The parameter list is a file which is read by the test-case driver programs in order to obtain biasing information which determines the type of pseudorandom sequences that the driver will issue. The parameter list interface object provided a convenient way for the driver programs to access the information in the parameter list file. It also provided a mechanism for the driver program to choose random entries from tables based on probability values.
- *Facility interface*
  The facility interface object provided a mechanism for a user to set or obtain signal and latch facility values in the event simulator, TEXSIM, or ZFS models. The facility interface object allowed the user to specify the model facility names in a separate parameter list file. If a facility name changed from one model to the next, the user updated the parameter list file for the new name, thus avoiding a program recompile.

The L2 chip simulation environment for the CMOS 4 S/390 processor contained test-case driver programs for the processor chips and for the BSN chips. These test-case driver programs were executed every simulation cycle and monitored the interfaces to the L2 chip. They provided stimuli into the L2 chip in a manner consistent with the interface protocol. The command stimulus issued to the L2 chip was based both on the bias values in the parameter list and on a random seed. The driver programs were enhanced to provide two modes of operation: heavy stress mode and random delay mode. In heavy stress mode, the test-case drivers monitored the L2 interface to determine which types of commands could be issued. Once this was determined, the drivers accessed the parameter list to choose from the subset of commands that could be issued. This mode of operation generally kept the L2 chip extremely busy, with few idle cycles between commands.

In random delay mode, the driver accessed the parameter list to choose a command first. If the command could not be issued because the interface protocol prohibited it, the driver would not issue any other commands until the chosen command was issued. This method of operation produced more gaps between commands and uncovered design problems which actually required a "less busy" state.

In addition to test-case driver programs, the L2 chip simulation environment contained automated checking programs. Like the test-case drivers, the checking programs were executed on each simulation cycle. They updated expected results dynamically, on the basis of events occurring in the simulation model. The automated checking programs ensured that data integrity was maintained in a multiprocessor system by interacting with the address space object to update the latest copy of the data when appropriate (for instance, when the driver program issued a store command to the L2) or by comparing data sent by the L2 to any processor against the expected latest copy of data. The automated checking programs also ensured that the ownership of each line in the address space was consistent with protocol. Any miscompares between the expected results from the automated checking program and the actual results caused the simulation to fail.

There were many other automated checking programs in this environment. Many of these verified that the L2 adhered to the interface protocols, that commands were processed in a timely manner, or that correct responses were sent from the L2 chip. Because the automated checking programs had no communication with the driver programs, a driver program could be removed and the associated automated checking program could remain in the environment. An example of this was the replacing of the BSN chip driver with the real BSN chip design. The L2–BSN interface protocol checking program remained in the simulation model in order to ensure that no interface violations occurred.

The L2 chip-level simulation environment was also used to simulate recovery scenarios. Errors were randomly injected into the L2, and the automated checking programs expected the appropriate recovery actions to be taken. After a successful recovery occurred, the simulation proceeded and the next random error was injected.

System verification

System verification of the CMOS 4 S/390 machine involved challenges not seen at the lower levels of verification. Components of the system design were implemented using multiple design languages. Methods had to be developed to compile the multiple languages into a single simulation model. Another challenge was in the area of resets, where different levels of code first come together. Finally, controlling the large model size so that the system can be run on the existing EVE 1.5 engines took finesse in swapping components for maximizing performance and coverage.

Mixed-language simulation Before an EVE 1.5 model was created, a ZFS companion model had to be created. This model was used for early system verification as well as for debugging miscompares that originally occurred on the EVE engine.

Previous S/390 designs were developed at a single laboratory where simulation models were described in a single database. The process for creating a two-component simulation model was also used for large-system models involving dozens of components (i.e., "macros," "units," or "chips"). With just a single design language, building a ZFS model involved linking components within the design-entry database, translating that single component into a flat "ZFS object," and then compiling a ZFS model from that ZFS object. Building an EVE model involved translating each design-entry component into a flat ZFS object and then to an "EVE object," linking the components' EVE objects, and then compiling the EVE model. In both cases, component objects were linked by name (i.e., as flat models) and not through a hierarchical description.

For the CMOS 4 S/390 system, nonlocal components were delivered as flat "TEXSIM objects," since those laboratories used TEXSIM for chip/unit simulation. The problem was to develop a process for building EVE (and ZFS) models which included those components. The components came from VHDL, BDL/CS, and DSL design description languages. One possible solution was to create system models as TEXSIM objects, but we found that this format was inefficient for large (five-million-gate) models, and was more suited for component simulation. The production-level solution to this problem involved changing the manner in which ZFS models were built. A "merger" was used to link ZFS objects in a hierarchical manner, preserving the names of component pins as aliases. A translator then converted TEXSIM objects into ZFS objects, and, with the construction of an appropriate hierarchical description, a single ZFS object was formed for the system model. That object was then compiled into a ZFS model in the standard way.

For EVE, the hierarchical description was used to affect the translation of (component) ZFS objects into EVE objects, allowing the existing EVE link-and-compile process to be used. This methodology of translating between simulator object-forms allows construction of a simulator-specific model from components described in any of a number of design-entry databases, thus avoiding a requirement that design communities adopt simulator-specific conventions.

Configurations The main limitation on the model configurations used for system verification was model size. System verification used both the EVE hardware accelerator and the ZFS cycle simulator. Model size was restricted by the amount of logic that EVE could support. The EVE model size capacity was determined by interactively adding chips to the model until the model outgrew the EVE capacity. ZFS, on the other hand, allowed larger models, but for debugging purposes the models had to match. By having matching models, failures on the EVE simulator could be played back on ZFS, where debug was more user-friendly. In order to use the system assurance kernel (SAK), an architectural verification program used to verify the hardware, the largest possible main memory space was required. This meant that the models must contain all four STC chips (or behaviorals). Along with the four STC chips, four BSN chips were necessary to support the function of the STC chips. Therefore, the chips that could be varied in the model were the processor, L2, and MBA.

From logic designers' input, it was decided that two L2 configurations would be most beneficial to test. The first was a logical L2 (two L2 chips), with all three processor chips attached. This placed the most stress on the L2 chips to ensure that they functioned properly with a heavy workload. The second configuration was one in which separate logical L2s were required to communicate with one another. This resulted in a model with two processors attached to two logical L2s. We then added two MBA chips to the "three-processor/two-L2" model **[Figure 8(a)]**, while the "four-processor/four-L2" model **[Figure 8(b)]** had no MBAs but used real STC chips. On the three-processor/two L2 model with the MBA chips, the STC behavioral was used, because the model size exceeded EVE capacity with the real STCs. This was not a concern, because the real STC logic would be tested on the other configuration. These two models provided the capability of testing all chips in the system (processor, L2, BSN, STC, and MBA), with a focus on the CMOS 4 chips (processor and L2).

One concern that arose from these two configurations was that the clock chip was not included with the other chips in the system. To address this concern, a two-cycle version of the chips was necessary [5]. The CMOS 3 (S/390 Parallel Enterprise Server G3) "nest chips" (BSN, STC, and MBA) were already in a two-cycle environment as a result of their DSL design language and compile process. The CMOS 4 processor and L2 chips were designed in VHDL, which allowed for smaller and faster one-cycle versions to run on our simulators. The two-cycle versions of the processor and L2 chips nearly doubled the size of these chips. In order to reduce the size of the two-cycle model, a single L2 chip model was built which provided a degraded version of the CMOS 4 system while still allowing the necessary clock testing. The resulting model (**Figure 9**) was one with three processors, half of a logical L2, two BSNs, two STCs, and one MBA.

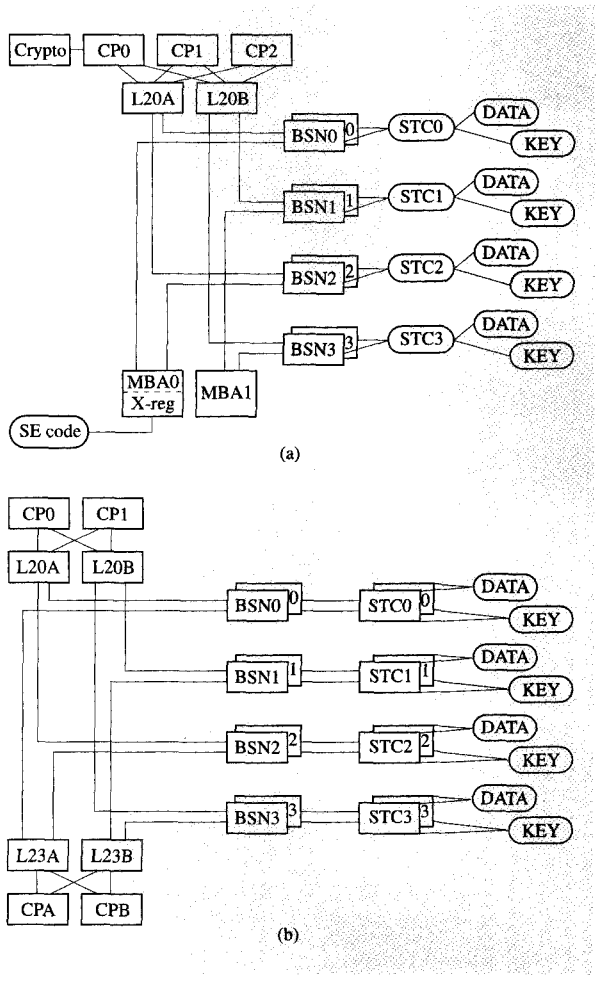

Figure 8

The two "one-cycle" system integration model configurations: (a) Three-processor/two-L2 configuration; (b) four-processor/four-L2 configuration. Boxes indicate real VHDL logic; ovals indicate behaviorals. Line connections represent both data and control lines.

Mainline SAK testing The mainline test strategy used for the CMOS 4 S/390 processor and L2 chips was similar to that used on previous S/390 machines [2]. SAK generated test instruction streams to verify the S/390 architecture and implementation of SMP system models running on the EVE 1.5 hardware accelerator. The EVE 1.5 enabled large-system models to run over 100 million cycles per week. For the CMOS 4 S/390 system, a new mapper

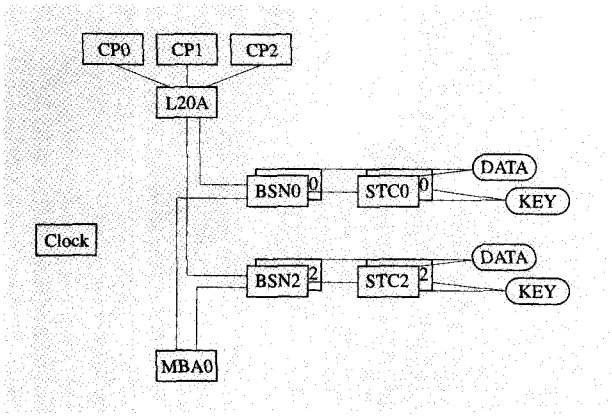

Figure 9

The "two-cycle" three-processor/one-L2 system integration model configuration. Representations as in Figure 8.

program (known as Memmove) was written in C to interact with the storage hierarchy utilizing the SimAPI interfaces [3]. Test-case specification parameters used by SAK were modified so that new architecture and specifics of the implementation were correctly handled during test-case generation.

All aspects of mainline test were done primarily by two individuals. This was accomplished by delaying SAK testing until the processor and L2 chips were verified by chip simulation to be functionally capable of working in the more complex system environment, rather than being bound by a development schedule that called for premature testing on the EVE machines. Once started, chip verification had completed most of the CP and L2 mainline testing and was concentrating on other aspects of the design. By staging the verification in this manner, mainline system test found the more complex problems rather than stumbling over simpler problems that should be uncovered at lower verification levels. This significantly reduced duplicate or concurrent problems and provided a more efficient use of the limited EVE 1.5 resources.

The need for system verification was underscored when two problems in the processor and L2 interface logic were discovered using the initial system-level models. However, after discovery of these bugs, millions of EVE cycles were run before the next bug was encountered. This bug was a complex SMP condition that involved three processors and back-to-back cross-invalidate (XI) requests. A tightly coupled relationship with chip-level verification tools and personnel and a graphical simulation trace browser helped reduce problem isolation time to a minimum. Furthermore, the designers typically turned around fixes in a few hours so that the fix could be verified and testing could continue in a timely manner.

Resets, IML One of the most visible metrics of the success of the verification effort comes when the system is powered-on on the engineering test floor. If the system is able to power-on-reset and start SAK, the design and simulation effort has achieved its first real measure of success. Therefore, it is important to verify the power-on-reset sequence prior to chip release.

Verification of power-on-reset on the CMOS 4 S/390 system presented a number of new challenges. The first was the use of the service word interface from the service element (SE) to the processor through the X-register in the MBA. This path is used to transfer data blocks, including millicode and I390 code [see the SE behavioral and X-reg connection in Figure 8(a)]. The second was the use of multiple levels of code to perform the subfunctions of power-on-reset.

One way to attack this testing which was used successfully in the past was to attach the real SE to the simulation model and drive it with the SE code [8]. However, this approach requires that the SE code be available early in the test cycle. Alternatively, the approach that was taken on the CMOS 4 S/390 project allowed the hardware to be verified without needing the SE code by using a state machine behavioral in place of the SE code. The behavioral controls the flow of the reset by sending and responding to service words on the X-register interface. This method enabled verification of the SE-processor communications and sequence through the power-on-reset. This sequence involved running through multiple layers of code. First, bootstrap millicode was loaded into the L2 cache via a fast load mechanism. The bootstrap millicode verified that the interfaces between the processors and the rest of the chips were functional. Next the functional millicode was loaded into main memory via the X-register. This code was then executed to set up the S/390 environment that is necessary for the next phase, when the I390 code is loaded into memory via the X-register. Finally, control blocks were built in memory and a final reset took place. Executing all of this code in a reasonable period of time required the EVE 1.5 hardware accelerator, which executed this environment at a speed of 300 cycles per second. At this speed, enough of the code was executed to ensure the verification of the hardware and the majority of the code layers.

This method of power-on-reset verification proved highly successful. Ten hardware problems and 35 millicode problems were found by verification. When the system powered-on on the engineering test floor, power-on-reset was achieved quickly, a significant achievement for a new processor design.

- *Explicit testing*

Clock testing

The clock chip was designed by the IBM Boeblingen Laboratory and was used to drive both CMOS 3 S/390 and CMOS 4 S/390 systems. Because of differences in the implementations of the processors, it was necessary to implement CMOS 4 S/390-specific functions in the clock chip. The CMOS 4 S/390-specific functions were verified separately by this team. The clock chip was driven by a behavioral developed for the CMOS 3 S/390 system and restructured to work in the CMOS 4 S/390 simulation environment. The clock chip was simulated on ZFS with test cases written in REXX using the SimAPI interface.

The functions verified on the clock chip were self-test, serial interface (SIF), single-cycle operations, chain shifting, and starting and stopping of clocks. Self-test on the CMOS 4 S/390 processor used the service element to control the initialization and signature checking, whereas the CMOS 3 S/390 processor used the clock chip to control and execute the entire self-test sequence. Testing in this area uncovered a multitude of design problems that were common to both processors. In the CMOS 4 S/390 design, the serial interface cycles only while the clocks are running and is inactive when the clocks are stopped. All valid commands for the serial interface were verified, with an emphasis on stopping the clocks during a frame transfer. Design failures were encountered on normal SIF operation, with two failures encountered on restart of the SIF after the clocks to the processor had been stopped. Single-cycle operations are similar on both systems, while stop on count end (SOCE) is a function unique to CMOS 4 S/390. This area required intense testing, with emphasis on the proper timing and interface protocol to the processor. This verification uncovered one timing problem on the interface. Chain shifting and starting/stopping of clocks were identical on both processors. This area was tested by driving a CMOS 4 S/390 processor and monitoring facilities and interfaces to ensure proper operation. The clock chip experienced a successful test-floor bring-up and functioned correctly with both of the systems. The test plan used to verify this chip will be used to verify follow-on clock chips in the S/390 family.

Array built-in self-test

Array built-in self-test (ABIST) test details can be found in a companion paper [5] in this issue.

Logic built-in self-test

A goal in design verification for this system was to double-check that the test patterns generated for chip manufacturing were correct. A prior methodology used a test-simulation model to run logic built-in self-test (LBIST) and generate a multiple-input shift register (MISR) signature that was compared to the actual MISR signature results on the new chips. Success was declared when the signatures matched on a cross section of the chips. Unfortunately, in this prior methodology, the signatures did not match in most cases, and a painstaking effort was put forth to analyze the differences between the simulation model and the hardware. With limited troubleshooting aids, it was very difficult to isolate the failure to an error in the modeling of the logic or a problem in the "real hardware." This process increased the test time on the chips and delayed the start of functional testing.

The new solution to this problem was to run LBIST on two independent simulation models and compare the signatures. If the signatures matched after a finite set of patterns were run, the probability increased that the MISR signatures generated by a set of test patterns were correct. To accomplish this goal, a two-cycle representation of the processor and L2 was created [5]. The two-cycle processor and L2 models were driven by the system clock chip. This model was initialized with the correct LBIST latch values and simulated using the ZFS cycle simulator. Each pattern required 4000 cycles on the L2 chip, while it took 10 000 cycles to accomplish the same task on the processor. The performance of these models ranged from 40 to 80 cycles per second. This pattern was then compared to the pattern generated by TestBench* [9]. When a mismatch occurred, the comparison usually showed that a latch was modeled incorrectly in either TestBench or ZFS. Latch modeling was then corrected on the failing simulator. The original goal was to get a minimum of ten patterns to match. The goal was exceeded, as more than 100 successful pattern comparisons were completed.

With this process in place, there was a high degree of confidence that the test patterns generated were correct. If a mismatch occurred between the simulated patterns and the real hardware on the testers, the problem was likely in the hardware. This expansion of verification helped reduce the chip test time dramatically, and chip test-pattern debug can now be completed in less than one week.

Scan-ring verification

The CMOS 4 S/390 system uses scan rings to reset the system rather than the logic-based reset used on prior systems. The system is forced to a reset state using a single scan ring per chip, with the scan data and controls arriving from the SE code through the clock chip interface. The processor and L2 have additional scan-ring capabilities through the use of LBIST, where the chip-long scan ring can be divided into 60 subrings. In LBIST mode, each of these subrings is used with random data patterns for hardware chip testing. Utilization of the subrings was an integral part of verifying the single long ring, as each

of the 60 shorter rings was simulated in parallel to create a faster chip-level scan verification.

The overall scan-ring verification effort had the following goals:

1. Ensure that every functional latch is on the scan ring and give latch count.
2. Ensure correct latch connectivity.
3. Identify all ring inversion points.
4. Determine design data (order of latches on the ring).
5. Verify scan starting/stopping capability.

Scan-ring verification was approached on four different levels. First, each macro was checked for connectivity by the individual designers. Next, the chip ring was verified using the event simulator. The third-level cross-checked the second-level test by running a Boolean check on the cycle simulator's software model. Finally, the chip scan ring was rotated in the middle of a mainline chip-level test case.

The first-level macro connectivity test was run on the event simulator. A generic C program was written to be used for all scan-ring macro testing. The program was personalized for the individual macros by a simple input file that defined the scan_in, scan_out, L1_clock, L2_clock, and scan_enable signals for the macro. After reading the personalization file, the program used the event-simulation software interface to traverse the model hierarchy and count the latches in the macro. Then, with the entire model in the "U" (uninitialized) state, scan clocking was applied and a pattern was scanned into the macro. The simulation was completed when the pattern appeared on the scan_out signal. The number of L1/L2 clock pulses required to scan the pattern through the macro was then compared with the latch count taken at the start. This testing uncovered problems with inversions on the ring, disconnected rings, and latches not appearing on the ring.

Chip-level scan-ring verification used a test scheme similar to that used by the macro level. The main difference was that the 60 LBIST subrings, each of length up to 1152 latches, were scanned in parallel because of event-simulator speed constraints (scanning the entire 60 000-latch processor ring would have taken about ten days). This test could not be executed on either cycle simulator because "U" data are supported only on the event simulator. A unique pattern was propagated through each subring.

The chip-level testing on the event simulator resulted in verification of macro-to-macro ring connections, global logical clock and scan control connections, and a chip-level latch count (derived from the sum of the length of the 60 subrings). The latch count was then cross-checked with the results from the third level of scan-ring verification: Boolean analysis of the two-cycle simulation model. Instead of actually running this model on a cycle simulator, the software tool traversed the model connections through the scan ring, flagging ambiguities or dual paths. The software also tracked the order of the latches on the ring, as well as the position of any inverters on the ring. This information was used to create the design data for initializing and debugging the hardware when it reached the engineering test laboratory.

The final scan-ring test used the cycle simulator to verify the system clocking in conjunction with scanning and scan clocking. A mainline test case was initiated with normal clocking. After running about half of the test case, the system clocks were stopped and the scan sequence began. Using the latch count derived in the second and third levels of scan verification, the ring was rotated with the scan_out pin connected back to the scan_in pin. When the scan operation rotated the ring exactly one time, the system clocks were restarted and the test case continued. A successful AVP test case indicated that the full rotation of the scan ring returned all latch values to a state identical to the one that existed before the clocks were stopped and that no arrays or combinatorial logic were erroneously clocked during the scan sequence. A single test case completed in six to eight hours, as the scan operation added 400 000 cycles to the test case. The test was repeated for a handful of AVP (processor) and random (L2) test cases.

This four-step methodology for verifying the scan rings was successful. No problems were encountered in any scan-ring connections or clocking, allowing the resetting of the system to proceed without hardware incidents.

When to release?

With the emphasis on time to market, a balance had to be struck between business and technical pressures. Therefore, the verification team strove for a clean first-pass system, with the understanding that success was measured by swift progress of the system through hardware systems bring-up. To achieve this, the logic design had to be able to perform most functions flawlessly.

The key to this strategy was threefold. First, the verification and design teams had to understand the strengths and weaknesses of the methodologies. The L2 random methodology, for example, is capable of uncovering very tight window condition error cases, but weaker at discovering static "hang" conditions where a small number of steady requesters lock one another out of the priority logic. With this type of information in mind, designers implemented hardware "dither" modes which can help break out of loops. At the same time, verification efforts to attack hang conditions were enhanced. Still, special care was taken to ensure that the dither mode would work if a hang condition were discovered in hardware system test.

Table 2 CMOS 4 S/390 logical bug totals.

	Designer macro level	*Unit level*	*Chip level*	*System level*	*Engineering test lab*
Problem count	1600	1400	1000	40	26
Percentage	39.4%	34.4%	24.6%	1%	0.6%

Second, priorities of hardware bring-up must be understood and fully verified. The hardware bring-up team's test plan was thoroughly examined by the verification team to understand the order in which tests would occur as well as the priority. Resetting the system at power-on was obviously a function that had no room for errors. Therefore, the entire reset sequence was fully simulated to give 100% confidence in the scanning and reset capabilities of the hardware. On the other hand, error injection, where a hardware test expert verified system recovery after a hard or soft error, was less important in the early stages of systems bring-up. An area such as recovery could afford to have a few problems found on the hardware and fixed in the second release.

Third, the work-around mechanisms that allow for avoidance of failing scenarios had to be understood and fully functional. Understanding the work-around mechanisms assisted in directing test cases toward the boundaries of the work-around conditions. This understanding also led to a test suite for the work-around mechanisms themselves.

With these principles in hand, comprehensive verification release criteria were created. The criteria were in checklist form so that each engineer could sign off on individual functions of the design. The criteria contained tests from all levels of verification. The release criteria were based on functional performance rather than number of simulation cycles, as in the past. While the functional performance measurement is currently qualitative, the verification personnel were best suited to make the judgment on how well the logic was performing. Although the number of cycles and error rate were used in the judgment, the most important factors were the completion of the test plan and the stress on the logic during simulation runs. Future release criteria will use coverage metrics to quantify the stress on the logic.

Results and concluding remarks

The success of the verification effort on the CMOS 4 S/390 system was judged at both qualitative and quantitative levels. While the stated goal of "swift bring-up and test of the hardware to aid time to market" is a qualitative statement, a quantitative hardware escape count has historically been used. Setting a quantitative target was therefore unavoidable. The number, based on the estimated maximum number of escapes that the engineering test laboratory could handle without affecting the schedule, was set at 40.

By both measures, the verification effort was a success. While only 26 hardware problems were found in the engineering test laboratory (**Table 2**), the main goal of "swift bring-up and test" was accomplished. The level of complexity of the escapes found in the laboratory was relatively high. However, all of the problems could be circumvented with relative ease, permitting testing to continue (**Table 3**).

All of the 26 hardware escapes were thoroughly analyzed by the verification team. The purpose of this analysis was to correct shortcomings in the process for this project as well as future programs. The analysis was separated into two parts: problem classification and methodology update.

The following process was followed for each escape analysis:

1. Escape is discovered in the engineering laboratory and assigned a problem number in a database.
2. The design and verification team debug the problem and reproduce it on a simulator. This step took from one to seven days, depending on the difficulty of reproducing in simulation.
3. The fix is verified in simulation.
4. The database is updated with a full description of the problem.
5. The verification team performs the analysis and classification.

Classification categories described information about the problem, the failure in the methodology that would have found the problem, the type of problem, the associated error in the design, the work-around capability, and the duration from problem discovery to understanding of the problem. These classifications helped define future methodology improvements and focus items for research and development.

The CMOS 4 S/390 logic verification results compare favorably to those for the previous CMOS 3 S/390 systems (50% fewer problems to the engineering test laboratory). Additionally, the results compare favorably to those for the previous nonderivative S/390 systems (one tenth of the

Table 3 Laboratory escape category and definitions.

Category	*Definition*	*Number of escapes in category*
Tolerate	Problem occurs rarely and is easy to recognize. A work-around is identified, but not used unless the problem becomes an annoyance.	10
Direct/nongating	The work-around fixes the exact occurrence of the problem and does not gate any further testing.	11
Indirect/some function disabled	The granularity of the work-around is such that other cases may take the work-around path or that some minor function testing may be gated.	5
None/major function disabled	No work-around exists, or the work-around causes major function testing to be gated or disabled.	0

problems found in the engineering test laboratory on the previous bipolar system). Much of the success in attaining the time-to-market goals for the system can be attributed to the verification methodologies used. The movement of verification engineers across multiple levels of simulation also contributed to the time-to-market success. The learning gained at the lower levels, along with the software tools that the engineers reused, were instrumental in quickly debugging higher levels and achieving targeted schedules. While future efforts will continue to benefit from new techniques such as formal verification, we expect that the methods adopted by our team will be used in conjunction with future development efforts.

Acknowledgments

The authors wish to acknowledge the contributions of tools and support personnel, including Ken Shepard, Tom Ruane, Gary Hallock, Dan Beece, Lisa Lacey, Rick Seigler, and Anne Huston. We also acknowledge the support and encouragement given by Paul Minear and Vijay Lund in their management roles.

References

1. A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "AVPGEN—A Test Generator for Architecture Verification," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* **3**, No. 2, 188–200 (June 1995).
2. D. F. Ackerman, M. H. Decker, J. J. Gosselin, K. M. Lasko, M. P. Mullen, R. E. Rosa, E. V. Valera, and B. Wile, "Simulation of IBM Enterprise System/9000 Models 820 and 900," *IBM J. Res. Develop.* **36**, No. 4, 751–764 (July 1992).
3. G. G. Hallock, E. J. Kaminski, Jr., K. M. Lasko, and M. P. Mullen, "SimAPI—A Common Programming Interface for Simulation," *IBM J. Res. Develop.* **41**, No. 4/5, 601–610 (1997, this issue).
4. B. Wile, "Designer-Level Verification Using TIMEDIAG/GENRAND," *IBM J. Res. Develop.* **41**, No. 4/5, 581–591 (1997, this issue).
5. Gary A. Van Huben, "The Role of Two-Cycle Simulation in the S/390 Verification Process," *IBM J. Res. Develop.* **41**, No. 4/5, 593–599 (1997, this issue).
6. D. K. Beece, G. R. Deibert, G. P. Papp, and G. F. Villanti, "The IBM Engineering Verification Engine," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 218–224.
7. A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator," *IBM Syst. J.* **30**, No. 4, 527–538 (1991).
8. S. Koerner and S. M. Licker, "Run-Control and Service Element Code Simulation for the S/390 Microprocessor," *IBM J. Res. Develop.* **41**, No. 4/5, 577–580 (1997, this issue).
9. W. V. Huott, T. J. Koprowski, B. J. Robbins, M. P. Kusko, S. V. Pateras, D. E. Hoffman, T. G. McNamara, and T. J. Snethen, "Advanced Microprocessor Test Strategy and Methodology," *IBM J. Res. Develop.* **41**, No. 4/5, 611–627 (1997, this issue).

Received December 9, 1996; accepted for publication May 19, 1997

Bruce Wile *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (bwile@vnet.ibm.com).* Mr. Wile is currently a Senior Engineer and Verification Manager in S/390. He has worked in verification since joining IBM in 1985, and was the verification team leader for the S/390 G4 (CMOS 4) system. Mr. Wile's previous verification experiences included storage controller element simulation for the S/390 bipolar ES/9000 machines including the 6-way, 8-way, and 10-way multiprocessor systems. He was previously verification team leader for the 10-way IBM ES/9000 system. Mr. Wile received a B.S. in computer science from Pennsylvania State University in 1984. He received an IBM Excellence Award in 1992 and an IBM Team Award in 1993, and in 1995 an IBM Invention Achievement Award for inventions and patent submissions pertaining to the TIMEDIAG/GENRAND tool set.

Michael P. Mullen *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (mmullen@vnet.ibm.com).* Mr. Mullen is currently a Senior Programmer; he joined IBM in 1976. He received a B.S. degree in computer science from Union College in 1976, and an M.S. degree in computer/information sciences from Syracuse University in 1981. Mr. Mullen has worked on the development of several mainframe systems, and is currently responsible for the hardware design verification of IBM S/390 CMOS processors. He received IBM Outstanding Technical Achievement Awards for his work on the IBM 3090 processor controller microcode (1985), ES/9000 processor simulation (1991), and AVPGEN development (1994).

Cara Hanson *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (carah@vnet.ibm.com).* Mrs. Hanson joined IBM in 1984; she is currently an Advisory Engineer. Since 1987, she has worked in the area of storage controller simulation for the IBM ES/9000 and S/390 G4 machines. Mrs. Hanson received a B.S. in electrical engineering from Rutgers University in 1984, and an M.S. in computer engineering from Syracuse University in 1994. In 1991 she received an IBM Gold Level Quality Award for her work on ES/9000 storage controller simulation, and in 1996 she received an IBM Team Award for her contributions to S/390 G3 common chip verification.

Dean G. Bair *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (dgbair@vnet.ibm.com).* Mr. Bair joined IBM in 1984 as a systems test technician. In 1986 he joined the IBM S/390 design verification team, where he is currently a Staff Software Engineer. He has worked on verification of I/O controllers, L1 cache designs, and shared L2 cache designs for the 6-way and 8-way IBM ES/9000 systems and the S/390 G4 system. In 1992 Mr. Bair received an IBM Outstanding Technical Achievement Award for his work on the 8-way ES/9000 BCE random test driver development. He received an IBM Invention Achievement Award in 1995 for his work on the TIMEDIAG/GENRAND tool set.

Kevin M. Lasko *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (km_lasko@vnet.ibm.com).* Mr. Lasko is currently an Advisory Engineer in S/390 simulation, performing system verification of the IBM S/390 G4 system. He joined IBM in 1978 in Subproducts Manufacturing Engineering. Since 1981 he has simulated various S/390 systems at the element and system level. Mr. Lasko received a B.S. in electrical engineering from Union College in 1978 and an M.S. in computer engineering from Syracuse University in 1983. He received an IBM Outstanding Technical Achievement Award in 1987 for his work on random test-case development for element simulation of the 3090 storage controller. In 1996 he received an IBM Team Award for his work on S/390 G3 common chip verification.

Patrick J. Duffy *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (pjduffy@vnet.ibm.com).* Mr. Duffy joined IBM in 1990 as a logic designer on an advanced processor design project. From 1992 until 1994 he worked on an IBM ES/9000 project as both a TCM coordinator and SCE designer. In 1994 he joined the S/390 design verification team, where he is currently a Senior Associate Engineer performing system verification. He received a B.S. in computer engineering from Lehigh University in 1990 and is currently pursuing an M.S. in computer engineering from Syracuse University. Mr. Duffy received IBM Team Awards for his design work on the ES/9000 project in 1993 and for his verification efforts on the S/390 G3 common chip in 1996.

Edward J. Kaminski, Jr. *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (eddiek@vnet.ibm.com).* Mr. Kaminski received a B.S. in electrical engineering from Rensselaer Polytechnic Institute in 1987, joining IBM that same year. He is currently a Staff Engineer, and has worked on verification of shared L2 cache and system controller elements of the IBM S/390 systems: 6-way, 8-way, and 10-way IBM ES/9000, and S/390 G4. Mr. Kaminski received IBM Team Awards for his work on the 10-way IBM ES/9000 in 1993 and S/390 G3 common chip verification in 1996, and an IBM Invention Achievement Award for his work on the TIMEDIAG/GENRAND tools in 1995.

Thomas E. Gilbert *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (tgilbert@vnet.ibm.com).* Mr. Gilbert joined IBM in 1974 and has held many technical and management positions. He is currently an Advisory Engineer in IBM S/390 design verification, working on the CMOS clock chip and system integration of the S/390 G4 system. His previous verification experiences include team leader and verification engineer in connection with various S/390 systems and CMOS channels. He has been working in design verification since 1984. Mr. Gilbert received an IBM Outstanding Innovation Award in 1988 for his work on I/O subsystem drivers, an IBM Outstanding Technical Achievement Award in 1990 for his work on scan-ring modeling, and an IBM Team Award for his work on the S/390 G3 common chip verification.

Steven M. Licker *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (slicker@vnet.ibm.com).* Mr. Licker is an Advisory Engineer working in IBM S/390 design verification. He joined IBM in 1977, and has held a number of technical and management positions in engineering systems testing on the IBM 308X and 3090 projects. Mr. Licker has spent the last ten years doing processor and system simulation on the IBM ES/9000 and S/390 G4 systems. He received an IBM Team Award for S/390 G3 common chip verification in 1996.

Robert G. Sheldon *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (rgs@vnet.ibm.com).* Mr. Sheldon is a Staff Programmer who, for the past twelve years, has worked on simulation accelerators (EVE) in support of IBM S/390 system simulation. He received an M.S. in computer science from Purdue University in 1976, and did postgraduate study at the University of California at Berkeley.

William D. Wollyung *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (bwollyung@vnet.ibm.com).* Mr. Wollyung joined IBM in 1974, and has worked in the simulation area since 1983. He is currently an Advisory Engineer working on storage controller simulation; he also worked on the hardware design verification of the IBM ES/9000 and S/390 G4 processors. In 1989 Mr. Wollyung received an IBM Outstanding Technical Achievement Award for his work on 3090 S simulation, and in 1991 he received an IBM Gold Level Quality Award for his work on the 9121 simulation team.

William J. Lewis *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (wjlewis@vnet.ibm.com).* Mr. Lewis joined IBM in 1982 and is currently an Advisory Programmer. He received a B.A. degree in computer science from the State University of New York at Oswego. He started in the hardware performance area working on workload characterization and storage hierarchy modeling. Mr. Lewis has been working on CP, storage, and I/O microcode verification since 1985, except for a brief return to performance to do CP modeling.

Robert J. Adkins *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (radkins@vnet.ibm.com).* Mr. Adkins is a Staff Software Engineer who is currently working on storage controller element simulation. From 1985 through 1995, Mr. Adkins was responsible for hardware design verification of the IBM ES/9000 and S/390 G4 processors. In 1991 he received an IBM Outstanding Technical Achievement Award for his work on ES/9000 processor simulation.