Functional formal verification on designs of pSeries microprocessors and communication subsystems

R. M. Gott
J. R. Baumgartner
P. Roessler
S. I. Joe

This paper discusses our experiences and results in applying functional formal verification (FFV) techniques to the design of the IBM pSeries® microprocessor and communication subsystem. We describe the evolution of FFV deployment across several generations of this product line, including tool and algorithmic improvements, as well as methodological improvements for prioritizing the portions of the design that should be considered for formal verification coverage. Improvements made in the formal verification toolset, including the introduction of semiformal verification and bounded-model-checking algorithms, have allowed increasingly larger partitions to become candidates for formal coverage. Other tool enhancements, such as phase-abstraction techniques to deal with clock gating schemes, are presented. Overall, numerous complex design defects were discovered using formal techniques across the microprocessor and communication subsystem, many of which would likely have escaped to the test floor.

1. Introduction

Functional formal verification (FFV) is the process of proving that a design adheres to its specification. Unlike simulation-based approaches, which may fail to expose certain design flaws, formal verification yields complete coverage with respect to the properties specified [1, 2]. The practical limitation of FFV stems from the fact that formal algorithms tend to require exponential resources with respect to design size. Thus, in practice, FFV can only be applied to the smaller design components. The task of developing a complete set of correctness properties to be verified at these lower design levels requires careful review by the design and verification teams. While FFV is guaranteed to expose all flaws with respect to the specified properties, design defects (commonly referred to as "bugs") may slip through the process because of omitted or improperly defined properties. Simulation environments, which scale to the chip and system level, benefit from the fact that at these higher levels even design flaws which are not targeted by dedicated checks are likely to propagate to other logic and ultimately be exposed. Thus, the exhaustive coverage for specified properties yielded by formal verification and the broader sampling of coverage yielded from simulation are complementary methods in the verification process.

Formal verification has been identified in the hardware development industry as a critical technology in the overall design and verification process because of its ability to expose design flaws that reside in rarely exercised paths. Such paths are probabilistically difficult to hit in a random simulation environment and can be challenging to hit in the fabricated hardware. Design defects that reside in these rarely exercised paths are referred to as "corner-case" bugs. These design flaws that remain unexposed throughout the pre-fabrication verification process lead to considerable expense in schedule delays once the problem is ultimately exposed on the test floor or in the field. Thus, industry has increasingly made use of formal verification techniques to catch such corner-case problems before fabricating the design. For example, bounded-model-checking techniques have been applied to the design of the Alpha

0018-8646/05/$5.00 © 2005 IBM

microprocessor memory subsystem [3], and Intel has invested in formal tools and methodologies to expose problems in its microprocessor designs [4, 5]. FFV has been deployed on the IBM pSeries* processors and communication subsystems since 1996 with the POWER3* chips. The POWER3 microprocessor verification effort used an early version of RuleBase [6], a formal verification tool developed by the IBM Haifa Research Laboratory, in an experimental and limited fashion. A larger formal effort was deployed on the POWER4* microprocessor [7]. Through the POWER4 effort, the verification team gained experience in using the formal tools as well as in choosing the most appropriate logic to verify. For the POWER5* verification work, improvements in the formal verification tools, including the availability of semiformal algorithms, allowed the verification team to test larger blocks of logic. This enabled formal verification to be deployed on more logic with less effort. The ability to test larger partitions allowed the team to target more encompassing architectural and microarchitectural properties, rather than the restricted subset of properties at lower-level interfaces. This additionally enabled the application of formal methods to tasks which hitherto would have been infeasible, such as the re-creation of test-floor failures and analysis of coverage results [8], both at larger design partitions encompassing numerous communicating blocks of logic.

This paper details the experience of deploying formal methods on the pSeries POWER5 processor and the communication subsystem, which consists of the pSeries High Performance Switch (HPS) and the Switch Network Interface (SNI). The motivation and value of using functional formal verification is discussed, as well as the strategy used in choosing where to deploy FFV within the designs. The processor and communication subsystem designs have differing design characteristics, which in turn affects the selection of the formal algorithms that will be most effective on those designs. Functional formal verification on the processor core posed unique challenges that were not encountered in the HPS or SNI designs. One primary difference in the processor design is the use of multi-phase latching schemes. The phase-abstraction technique described in Section 5 details the methodology developed to deal with this complexity, which helps make FFV feasible on such a design. A second difference between the designs is the type of partitioning appropriate to the designs. Within the communication subsystem, meaningful partitions could be readily identified that fit within the size limitations of FFV. Specific properties within these designs could be difficult to fully prove at times, but some level of meaningful coverage was generally attainable. In the POWER4 timeframe, a common challenge on the processor core was simply to identify a meaningful partition of logic that was within the size limitations of FFV. While numerous design defects were found that were significant to the integrity of the POWER4 design, the FFV deployment was often restricted to lower-level partitions that could fit within the size limitations of the tools, rendering the overall architectural and microarchitectural coverage attained through those efforts somewhat lacking. The introduction of semiformal methods in POWER5 helped alleviate these restrictions.

The remainder of this paper is organized as follows. In Section 2 we describe the overall system under verification. The motivation for FFV described in Section 3 and the general FFV strategy described in Section 4 are common to the experience of applying FFV on the HPS, the SNI, and the processor core. The phase-abstraction techniques described in Section 5 were developed in response to the unique challenges of the processor core. Similarly, the development of the semiformal verification methodology, described in Section 6, was motivated largely by the challenges of applying FFV to the POWER4 processor core. Section 7 describes the FFV tools used in these efforts, and Section 8 describes the results. In Section 9, we summarize the work presented in this paper.

2. Design overview

The capabilities of the IBM eServer* p4 and p5 systems are extended through the use of the pSeries High Performance Switch (pSeries HPS) communication subsystem [9]. The eServer p4 system is based on the POWER4 microprocessor, and the eServer p5 system is based on the POWER5 microprocessor. The POWER5 microprocessor was introduced in 2004 and is a derivative of the POWER4 microprocessor design [10]. The POWER5 microprocessor design features two-way simultaneous multithreading (SMT), enabling the processor to fetch instructions from more than one thread. The POWER5 implementation allows instructions to execute from each thread when possible, and allows instructions from one thread to utilize all execution units if the other thread encounters a long-latency event. Fully verifying the SMT functionality was a key verification challenge on POWER5; hence, much of the formal verification resource was devoted to verifying aspects of SMT. The total switching power of the microprocessor core increased with the introduction of SMT as more instructions executed per cycle. To help offset this, the POWER5 design implements aggressive dynamic power management to reduce the switching power. The design uses a dynamic clock-gating mechanism to gate off clocks to a local clock buffer if dynamic power management logic detects that the set of latches driven by the buffer will not be used during the next cycle. The dynamic clock-gating implementation

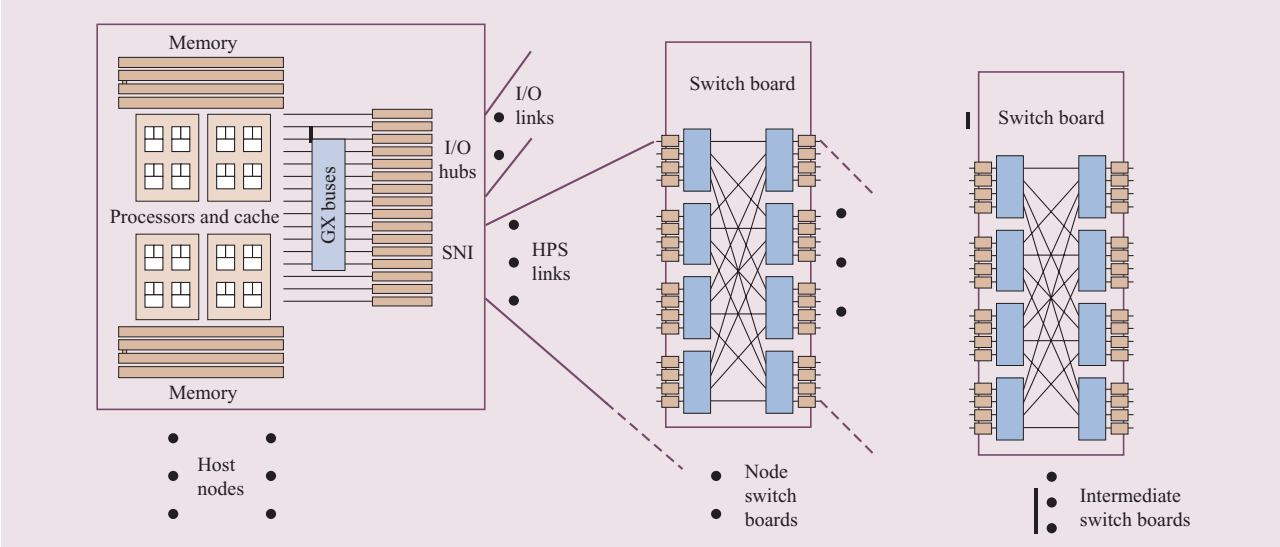

Figure 1

HPS SAN topology.

added complexities to the verification process, which is described in Section 5.

The HPS System Area Network (SAN) architecture is designed for efficient interconnection of processing and I/O nodes. The HPS SAN was first introduced with the eServer p4 system, providing large-scale system performance to the 50–200-Tflops range. Its high link bandwidth, robust network scalability for high throughput to thousands of endpoints, and support for multiple classes of traffic types make it suitable for a wide range of uses in high-performance server systems. The SNI provides connectivity between separate nodes over the system area network to form a message-passing cluster. The HPS chip is a hardware element that routes traffic from one node to another node, designed such that any node can connect to any other node in the system. The SNI, also known as the communication adapter, is a microcoded hardware element that is a physical interface between the node and the switch. The SNI and the HPS chip enable the exchange of information at a very high data rate among a large number of high-performance SMP nodes. Each SMP node connects to an HPS switch board through an SNI physically contained within the node itself, as shown in **Figure 1**. The SNI was updated for the eServer p5 system to allow for connection over a GX+ bus. The same pSeries HPS design serves both p4- and p5-based systems.

3. Motivation for FFV

Since the advent of POWER4, investment has been made both in tools development and in execution, with the ongoing goal of increasing the quality of the hardware designs. This investment has included a significant focus on FFV owing to the following types of benefits obtained by those technologies:

- Executing FFV on a design tends to result in a reduced number of test-floor escapes to (refer to the subsection on re-creation of test-floor failures).
- Complex design defects tend to be discovered earlier in the verification cycle, enabling more thorough and robust fixes to be applied without risking a schedule slip. Generally, the random simulation process first exposes "simple" defects—those exposed by relatively simple input patterns. Complex bugs are design flaws that require a more stressful input stimulus to expose. It can take random simulation running millions of simulation cycles over the course of weeks or months to achieve the necessary coverage to expose complex bugs.
- A higher-quality and more robust design often results from the identification of performance bugs, "unexpected interface assumptions" required by the design to ensure its functional correctness, and other subtle problems that are discovered through the FFV process.
- FFV tends to increase the understanding of the design for both the designers and the verification team, improving the quality of design documentation.
- Regression is superior in formal verification. FV regressions are typically completed within hours or days. In simulation, it can take months to achieve

coverage goals that give a satisfactory level of confidence in the regressed model.

The types of design defects found by FFV can generally be categorized as one or more of the following:

1. *Schedule advantage:* Formal verification can generally begin earlier than higher levels of verification, such as unit or chip simulation. For example, FFV may be performed on a partially implemented design as soon as it compiles. Thus, the discovery of some design flaws by FFV early in the verification cycle is simply due to the fact that FFV was the first to evaluate the logic. The resolution of this class of design defects helps provide a more stable model for subsequent simulation, and enables a more robust design to be developed as bugs are flushed out directly as logic entry is performed.
2. *Exposure of gaps in simulation coverage:* FFV may reveal coverage gaps in the simulation environment. For this class of design defects, it is often deemed important for simulation to add the appropriate modifications to driver/checker code so that simulation can expose the failing behavior as well.
3. *Corner-case and complex bugs:* FFV can expose corner-case bugs that are extremely difficult to find in a random simulation environment. These represent the true power of FFV to expose problems that are too probabilistically difficult to expose with simulation alone, that likely would be incorporated in the chip design (commonly referred to as "slipping into silicon") without formal analysis. Complex bugs, as discussed above, are design flaws that would likely be exposed in simulation once a certain level of coverage was achieved. The exhaustive nature of formal verification allows such problems to be exposed earlier in the verification process.
4. *Performance flaws:* FFV can expose design defects that would manifest themselves as performance problems in the hardware design rather than functional problems. This is an important class of design defects, because it may be exceedingly difficult to detect performance-related problems through simulation, and very difficult to resolve them if they are observed only on the test floor. Especially in the area of high-performance computing (HPC), performance is function, and performance-related problems can be a reason to re-fabricate the design.
5. *Spurious failure:* Some defects exposed by FFV may be deemed "impossible" in the current environment. That is, the neighboring logic could not behave in a way that drives the failing input sequence to the block interface. Nevertheless, the designer may often consider such a problem to be a true bug exposing an "unexpected interface assumption" required by the design to ensure its correctness, and provide a fix. Such fixes are often beneficial, since the behavior on the interface may occasionally change at some future point in time and possibly exhibit the previously disallowed behavior, resulting in a true hardware flaw. Overall, such changes tend to increase the rigor of the design.

4. Strategy

Two important aspects of the formal methodology are the selection of the design portions to verify and the development of the formal specifications used to verify those portions. In this section we address these topics.

Choosing logic for FFV

Not all logic that could be verified with FFV is actually verified that way. The formal verification team works with the simulation and design teams to identify logic that would most benefit from formal coverage. The initial lists of possible FFV candidates are then prioritized and refined to a smaller list, which represents the logic deemed most critical. As experience is gained with formal methods, a set of guidelines on choosing logic for FFV has evolved. These guidelines include the following:

- Logic that is difficult to stress through random simulation. Logic that falls into this category often has many independent interfaces and hence is prone to corner-case bugs. It is difficult for random simulation to fully exercise all possibilities of inputs on those interfaces, risking late defect discovery and incorporation in the chip design.
- Logic that is difficult to verify in simulation, and for which correctness is essential to the overall design integrity. Some properties, such as the absence of deadlock[1] or livelock,[2] are nearly impossible to verify in a simulation environment. Formal verification is able to prove the correctness of such properties (if the model size permits); such confidence cannot be gained through simulation.
- Logic that was problematic in previous versions of the design. Some portions of the design are evolutionary in nature. If there was a test-floor escape in a particular portion, that logic is likely to be considered for proactive FFV in the next version of the design.

[1] *Deadlock:* A condition that occurs when two processes are each waiting for the other to complete before proceeding. The result is that both processes fail to complete.
[2] *Livelock:* A condition that occurs when two or more processes continually change their state in response to changes in the other processes. The result is that none of the processes will complete.

- New logic (i.e., not obtained by a simpler evolution from a prior design) that is anticipated to be complex and error-prone. For such complex new logic, FFV is considered a strong mechanism to help stabilize that logic more quickly.
- Logic that is not covered by simulation until late in the verification schedule. Simulation sometimes cannot be used to cover all logic in the desired timeframe. In such cases, FV might be used to help provide coverage.

Developing the specification

After a partition of logic has been chosen for formal coverage, two elements must be determined: a) how to drive legal stimulus to the interface, and b) what properties to verify. Discussions with the design and simulation team are held to help determine how to model these specification aspects. As discussed in the subsection on strategy for choosing between formal and semiformal approaches, properties to verify tend to fall into the following three categories, which may affect the choice of formal tools and algorithms:

- "Sanity check" properties.
- Liveness properties.
- Functional correctness properties.

"Sanity check" properties are simple invariants that describe a good event that should always be true, or an illegal event that should never occur—for example, "these multiplexor selectors must be one-hot," and "this state machine should never be in more than one state." In many designs, this type of property holds true even with completely random behavior of the driver, and it tends to be very simple to code. When FFV is begun on a new piece of logic, such properties are often good to verify first. They can be extremely powerful in exposing complex design defects, and have the benefit of being straightforward to code and maintain, while offering design insight that helps the development of the other, more-encompassing types of properties.

The second type of property, a liveness property, describes a desired or necessary system condition that must eventually be reached [1]. This is a key area for formal methods, since it is nearly impossible to verify such correctness in a simulation environment. An example liveness property is the following: "when a *request* is received, the design must eventually issue an *acknowledgment*." Unbounded liveness properties tend to be simple to code, and can be extremely powerful in exposing complex bugs. However, unbounded liveness properties tend to be much more computationally expensive to verify than the other classes of properties. For this reason, on larger design partitions, it is occasionally beneficial to attempt to cast an *unbounded* liveness property as a *bounded* liveness property. An example of a bounded liveness property is the following: "when a *request* is received, the design must issue an *acknowledgment* within 20 clock periods." Most unbounded liveness properties may be straightforwardly translated to bounded liveness properties in this manner. The only practical difficulties we have encountered in such a translation are due to the presence of "fairness" constraints needed to enable a liveness proof. A fairness constraint is one that specifies that a certain condition must occur infinitely often. For example, the liveness property "every low-priority *request_lo* eventually receives an *acknowledgment*" may require the fairness condition "the higher-priority *request_hi* is inactive infinitely often." Note that fairness conditions often imply a strategy to translate more complex liveness properties into bounded variants, such as "when a *request_lo* is received, the design must issue an *ack* within 20 clock periods as long as *request_hi* remains inactive." Translating unbounded liveness properties into bounded ones requires manual effort, though the bounded properties have the benefits of lesser computational complexity to verify; being suitable for a wider range of verification algorithms (such as semiformal algorithms); and yielding more performance insight than mere liveness properties alone.

The third class of properties, the functional correctness properties, verify that the implemented design behaves as dictated by its specification. These properties tend to capture higher-level behavior—for example, "in response to a particular sequence of events, the design produces a correct sequence of responses." These properties are generally more difficult to code and maintain. Often, a reference model that at least partially mimics the predicted design behavior is needed to verify functional correctness. Compared to the "sanity checks" and liveness properties, the functional correctness properties tend to have a greater number of "spurious failures," in which the property or driver is incorrect rather than the design. The higher number of false failures often stems from the design specification being inadequate, or unexpected "special-case" situations that have to be specifically accounted for. In spite of the extra burden associated with functional correctness properties, they are a valuable class to include in the verification effort, often necessary for capturing architectural or microarchitectural correctness. They are capable of uncovering complex design flaws and exposing performance-related design defects. The performance-related bugs discovered through this type of verification are extremely valuable in that such problems are often difficult to detect and debug in a simulation environment.

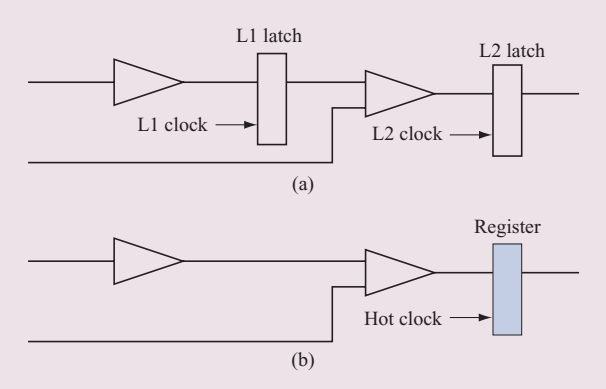

Figure 2

Phase abstraction: (a) Two-phase level-sensitive latch-based design. (b) Corresponding phase-abstracted register-based design.

5. Design modeling

All design modules were written in Very high speed integrated circuit Hardware Description Language (VHDL) and converted into a netlist representation using a VHDL compiler. All memory arrays (e.g., RAM blocks) were expanded into a two-dimensional grid of state elements called *registers*. The other state elements in the processor core were designed using primarily level-sensitive latches to help enable the degree of pipelining needed for the high clock speeds therein. The technique of phase abstraction was utilized to automatically convert this logic into a more compact register-based representation, as described below. The communication subsystem was directly designed using registers.

The multi-phase latching scheme pervasive in the processor core had posed challenges to formal verification since POWER4 for several reasons. First, most formal algorithms rely upon enumerating the reachable states of the design under verification, which generally requires exponential resources with respect to the number of state elements in the design. Multi-phase latch-based designs tend to comprise a significantly greater number of state elements than functionally correspondent register-based designs. Second, the depth of such multi-phase designs—i.e., the number of verification timesteps necessary to enumerate all reachable states—tends to be several times greater than that of functionally correspondent register-based designs. This is because an oscillating clock must be modeled that sensitizes only a single phase of latches per verification timestep, whereas a register-based design may be evaluated at a full clock period per verification timestep. This also tends to increase the amount of computational resource necessary to obtain a given amount of coverage using formal and semiformal verification.

Phase abstraction [11] is a commonly used technique for enhancing the coverage attainable with formal and semiformal verification on multi-phase designs. Phase abstraction converts latches of all but a single phase to wires, and converts the remaining phase into registers. This transformation is depicted in **Figure 2** for a two-phase design. Figure 2(a) depicts the original design, with latch phases shown as L1 (level 1) and L2; the L1 latches are clocked by an L1 clock, and the L2 latches are clocked by an L2 clock (which is often modeled as opposite in polarity to the L1 clock). Figure 2(b) shows the phase-abstracted design, in which the L1 latches are converted to wires and the L2 latches are converted to registers that are "hot-clocked"[3] every timestep. More formally, the resulting registers sample the data at their inputs every timestep and re-drive that sampled data to their outputs one timestep later; an oscillating clock is no longer used to prevent sampling at certain timesteps.

While this phase-abstraction technique was initially developed for POWER4, the aggressive power savings utilized in POWER5 resulted in a significantly larger diversity of clocking and latching schemes, including "clock gating," whereby one may utilize functional logic to dynamically turn the clocks on and off to portions of the design. Such clock gating may clearly affect the functional correctness of the design; if the clock used by a portion of the design is turned off at the wrong time, it may yield incorrect computations. The existing phase-abstraction methodology thus had to be made more flexible to enable classification of the various clocking schemes and identification of the clock-gating schemes.

Our solution to this problem was to develop a new methodology for annotating the HDL of the local clock buffers to indicate whether they were gated or nongated; in the former case, we also annotated the functional "clock-gating" signals that were used to turn off the individual clocks. We next developed an automated toolset that interpreted these annotations and performed phase abstraction accordingly. This automated toolset is integrated into the design importation path in the formal/semiformal verification toolsets. The transformation performed by the phase-abstraction toolset for non-gated latches is as shown in Figure 2. The transformation of gated latches differs; such latches are first transformed into semantically equivalent non-gated variants, then abstracted as in Figure 2. This preprocessing transformation is depicted in **Figure 3** for a gated L1 latch. The GATE signal is removed from the clock expression to the L1 latch, enabling the L1 to be directly clocked by a non-gated L1 clock. The GATE expression selects a newly created multiplexor; when "1" (i.e., the GATE is not disabling the latch clock), DIN (data in)

[3]Hot-clocked: Clocked by an always-active clock rather than by an oscillating clock.

combinationally drives the input to the L1 latch. When "0," the L1 latch input is driven by a newly created L2, used to hold the last-driven value of the L1 latch (or its initial value, at time 0). With this transformation, all resulting functional clocks may be held active (since the gating is reflected by data feedback loops); all nonfunctional (e.g., built-in self test) clocks are held inactive during the functional verification.

Once these modifications to the methodology were in place, the local clock buffers were easily annotated to enable phase abstraction to identify the various clocking schemes and perform the abstraction accordingly. The formal and semiformal efforts were consequently able to exploit the coverage benefits of phase abstraction despite the more aggressive dynamic power-management scheme.

6. Semiformal verification

The term *semiformal verification* refers to a hybrid type of design exploration that moves iteratively between random simulation and a resource-bounded exhaustive search. The simulation portion is able to quickly reach deep states, and the bounded exhaustive search is able to reach probabilistically difficult states. This approach avoids the state–space explosion issue encountered in a pure unbounded exhaustive search, yet yields much higher coverage than a standalone simulation run [12]. This process is depicted in **Figure 4**. The red circles indicate the exhaustive bounded search of the state space of the design, and the zigzag lines indicate random simulation. The depth of each exhaustive bounded search is controlled by some resource bound, possibly a time limit or a depth limit. The depth of each random simulation run is also bounded by a resource limit. Each semiformal iteration consists of a bounded exhaustive search phase and a random simulation phase. If any design defects are exposed during this process, they are reported. Otherwise, the semiformal engine selects a state encountered during its previous processing and seeds the next iteration into it. The next phase of the exhaustive search and random simulation is started from this seeded state. This choice is often made using some type of coverage analysis to select a state which seems the most different from previously encountered states, or one which heuristically seems the closest to a bug.

As depicted in Figure 4, the search begins from the designated initial state of the design at the bottom left. Two bugs are shown in the design; one is reachable rather shallowly from the initial state of the design (at the left), and the other is reachable only very deeply from the initial state (at the right). The red circles surrounding the initial state indicate bounded-model checking. Note that the initial bounded-model check comes very close to hitting the leftmost bug. If given more resources, this search likely would have generated a "shortest trace"

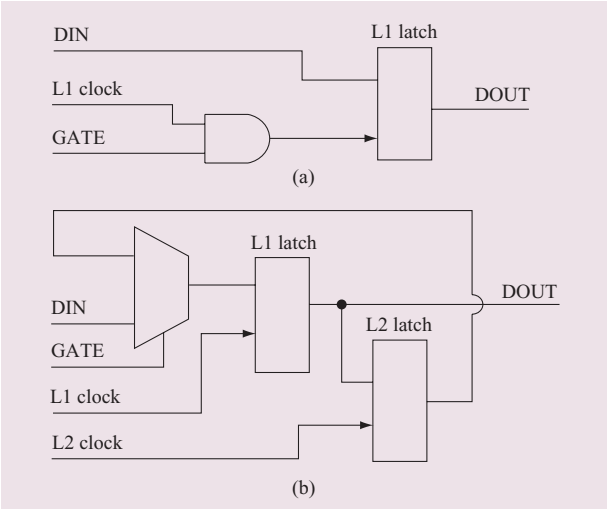

Figure 3

Transformation of gated-clock latches: (a) Original gated-clock latch. (b) Semantically equivalent non-gated model.

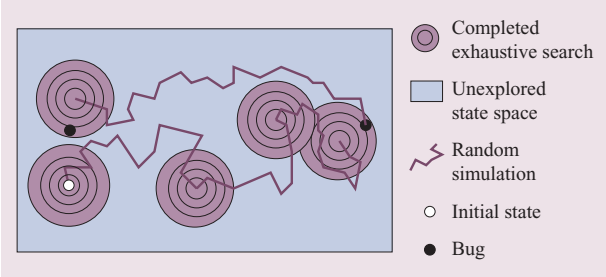

Figure 4

Illustration of semiformal search.

exposing this bug (though note that exhaustive bounded search generally requires exponentially increasing resources as its depth increases, often rendering it unable to detect deeper design defects on larger designs). As depicted, numerous semiformal iterations are performed before this bug is found, in this case yielding a much longer trace than necessary. Overall, the shorter exhaustive-search-only traces tend to be much easier to debug for several reasons. First, they contain less information because of their shorter length. Second, the formal algorithms tend to illustrate only "necessary" toggling of the design logic to expose the design defect, whereas random-simulation-based traces tend to contain a large degree of unnecessary activity.

For deep bugs such as the one to the right, random simulation and semiformal verification are often the only approaches to expose such flaws; exhaustive search alone

fails to penetrate deep enough on larger designs. The motivation for using semiformal verification instead of random simulation alone is generated by the immense increase in coverage achieved by the bounded exhaustive search in semiformal verification.

Methodologically, it is often beneficial to apply several types of semiformal search in parallel on a given design. One type performs many semiformal iterations, allowing relatively little resource for each individual exhaustive search and simulation search per iteration. This type of run is useful to quickly obtain relatively high coverage throughout the state space. Another type devotes large resources to each exhaustive search phase, allowing the exhaustive search to proceed as deeply as possible to help obtain short traces and to obtain higher coverage local to the semiformal-seeded states. With respect to Figure 4, the former type of application correlates to more yet smaller red circles spread throughout the state space, whereas the latter correlates to fewer yet larger red circles spread throughout the state space.

In random simulation frameworks, the notion of "random biasing" is critically important to ensure the highest coverage. Semiformal verification is less dependent on random input stimulus, since it requires only that simulation come relatively near a bug, then may use exhaustive search to fill in the gap and locate it. Nevertheless, for highest coverage, semiformal verification is still dependent upon random simulation to provide a good sampling of the state space—i.e., to touch states that are at least "near" to all reachable states—to help ensure that no design defects are missed. Random biasing of certain types of inputs is thus practically important in a semiformal verification methodology. For example, a design may have a *flush* input which, if asserted, will bring the design back to the region of its initial state. It may be necessary to toggle this input occasionally to ensure that all design flaws are exposed, but this may be required only very infrequently (e.g., once per 1,000 timesteps rather than once per two timesteps on average). The biasing of such an input may thus be adjusted to prevent the random simulation from effectively getting stuck near the initial states, in turn reducing semiformal coverage of deeper states. Note that the exhaustive search itself ignores all biasing by definition.

7. Formal verification tools

Two internally developed formal verification tools, RuleBase and SixthSense, were used in the POWER5 verification. RuleBase [6] was developed by the IBM Haifa Research Laboratory and initially released in 1996. At the core of the original RuleBase was an enhanced version of Symbolic Model Verifier (SMV), a symbolic model checker licensed from Carnegie Mellon University [13]. RuleBase has had a number of enhancements that were delivered in the POWER5 timeframe, which are detailed in the next section. Midway through the POWER5 verification cycle, the internally developed IBM SixthSense semiformal verification tool was released. The release of SixthSense allowed the verification team to target blocks of logic that were too large for pure exhaustive search techniques. RuleBase and SixthSense are described in the next sections, followed by a section detailing the strategy for choosing a particular tool for given verification problems.

RuleBase

In the POWER5 verification timeframe, the IBM Haifa Research team made available to internal users a beta release of the RuleBase Parallel Edition (PE) [14]. RuleBase PE, which became publicly available in February 2004, offers many enhancements over the original version of RuleBase. In RuleBase PE, additional verification engines are made available to the user. RuleBase PE dispatches the selected verification engines to run on the same problem in parallel. The results of the multiple engines are coordinated such that if one engine finds a failure or proves the property, the other engines are halted. This allows the user to dispatch multiple engines, or even the same engine with different runtime parameters, to work on a single problem. In general, the user does not know which engine may perform the most efficiently for a given problem. Thus, dispatching multiple engines can lead to savings in verification time. The verification engines that were used in the work reported in this paper are briefly described here:

- *Discovery:* Discovery is an optimized SMV-based model checker. Safety properties are evaluated on the fly during computation of the reachable states [14]. A safety property describes a condition that must not be violated at any instance in time. For other properties (e.g., liveness), the reachable states are used to simplify the model before classical model-checking algorithms are deployed to perform the verification. Discovery can evaluate multiple properties in a single run, and can find counter-examples (cases in which the property has been violated) as well as prove properties.
- *Satisfiability engine (SAT):* SAT is a bounded-model-checking engine that performs bounded exhaustive search from the designated initial states of the design in an attempt to find a case in which the property under test does not hold (refer to the initial exhaustive search in Figure 4). SAT can verify only a single property at a time (safety only), and cannot generate proofs. The SAT engine in RuleBase PE is an

enhanced version of the Princeton Chaff SAT engine [15].

- *Unfolding:* The Unfolding engine is a Boolean Decision Diagram (BDD)-based bounded-model checker with under-approximation support to enable deeper yet incomplete searches [16]. Unfolding attempts to find violations of safety properties, and may evaluate multiple properties in a single run, but generally cannot yield proofs.
- *Beelzebub:* Beelzebub is a BDD-based adaptive search engine combining forward and backward search as well as various approximations. It operates on a single safety property at a time and is most effective at finding fails, though it is also able to complete proofs [17].

RuleBase also incorporates several reduction techniques that automatically reduce design size before the above verification engines are called, which reduces their degree of exponential blowup. RuleBase utilizes a set of unique languages: the Environment Description Language (EDL) for writing the driver for the HDL logic, and Sugar [18] for writing properties. A more complete discussion of EDL and Sugar can be found in [19].

SixthSense

SixthSense [20] is a semiformal verification toolset that was developed in the IBM Systems and Technology Group, overlapping with the development of the IBM combinational equivalence checker Verity [21]. SixthSense is a novel transformation-based verification toolset; it integrates numerous automatic complexity-reducing abstraction algorithms to iteratively and synergistically simplify and decompose complex problems into simpler ones that may be processed more easily by terminal verification algorithms. Example abstractions range from straightforward reduction techniques, such as constant propagation and redundancy removal, to more novel and aggressive transformations, such as minimum-area (min-area) retiming. The incorporated verification algorithms range from random simulation to pure formal analysis to a hybrid semiformal search that iterates between random simulation and a resource-bounded exhaustive search, as discussed in Section 6. The primary engines that were used in the work reported in this paper are the following:

- *Semiformal search:* This engine performs hybrid semiformal search that moves iteratively between random simulation and exhaustive bounded search (primarily using the SAT engine described below). This engine is the primary bug finder of SixthSense.
- *SAT:* SixthSense uses the structural SAT solver shared with Verity, which integrates SAT algorithms with structural simplification techniques and BDD-based analysis [22]. The robust fusion of these techniques enables applications for deep bounded analysis under semiformal search, even on large designs.
- *Redundancy removal:* There are two engines in SixthSense that perform redundancy removal. One performs simpler combinational analysis on sequential designs to eliminate constant and functionally equivalent gates [22]. The other performs more aggressive sequential analysis (e.g., induction) to achieve a similar type of reduction; it requires greater computational resources, though it yields greater reductions than the first engine. The latter engine is also useful in providing quick proofs of correctness in certain cases, even on very large designs.
- *Min-area retiming:* This engine applies the design technique of retiming (relocating registers beyond combinational gates) to reduce the total number of registers in the design [23]. Particularly on the highly pipelined processor core, retiming tends to be a very powerful reduction technique, capable of yielding 50% or more register reductions than would be possible through redundancy removal alone. The synergy between retiming and redundancy removal tends to yield incrementally greater reductions through repeated applications.
- *Reachability:* This engine evaluates the reachable states of the design to enable proofs. As with any reachability engine, this tends to be the only size-limited engine used in SixthSense, prone to memory-outs beyond several hundred registers.

SixthSense uses a VHDL-based specification language called BugSpray. BugSpray is used both for specifying safety properties to check for the design under verification and for specifying the random drivers that govern the behavior of the input stimulus applied to the design under verification. The Bugspray language is presented in more detail in [24].

Capacity of tools

The capacity of the formal verification toolset has increased significantly since the POWER4 verification work. At that time, RuleBase, the primary formal verification tool, was capable of exhaustively verifying design partitions of up to approximately 300 registers after reductions [6]. RuleBase could load initial designs (before automatic reductions) of up to a few thousand registers. Enhancements to RuleBase since then have allowed for much larger initial designs to be loaded

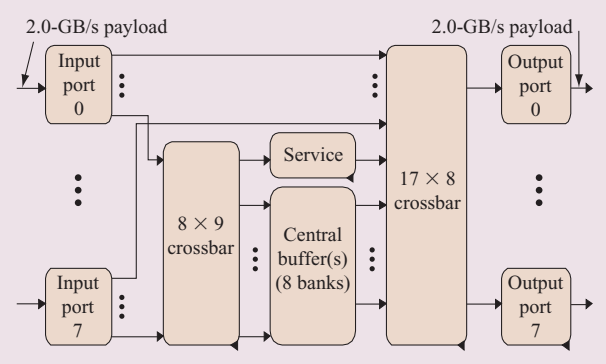

Figure 5

High-level view of the pSeries HPS chip.

(in the ten-thousands of registers). SixthSense became available during POWER5, and can load initial designs with 100,000+ registers; the largest model imported into SixthSense (though not on this project) had several million registers. Nevertheless, note that formal algorithms generally require exponential resources with respect to design size, particularly register count. Thus, as the size of the reduced design increases, the likelihood of completing a proof with either tool decreases; semiformal verification and bounded-model-checking algorithms may then be the best mechanisms to perform bug-hunting for larger designs.

The introduction of bounded-model checkers and semiformal verification provided the ability to work with increasingly larger design partitions; this allowed efforts to be dominated by the type of architectural and microarchitectural properties for which verification was desired, instead of by the size of the design portions that could be fitted into the formal tools. While full proofs may not always be obtained, these techniques have proven to be powerful bug finders, able to expose complex corner-case problems that would be difficult to uncover using random or directed simulation approaches alone. Verified block sizes are listed for the HPI, SNI, and processor in Section 8.

Strategy for choosing between formal and semiformal approaches

In verifying a portion of the design, the strategy used in choosing between formal and semiformal approaches has been governed largely by the size of the reduced design. If the reduced design contains more than 400 registers, it is highly likely that a semiformal or bounded-model-checking effort will be the best approach, since the chance of obtaining a proof may be diminished. In other words, as the size of the reduced design increases, the verification goal often moves away from obtaining full proofs to exposing design flaws. In some cases, it is known at the beginning that the design size is likely to be too large for an exhaustive effort. In those cases, a semiformal approach is taken from the start; obtaining proofs on such larger designs is desirable but is not deemed as critical as obtaining high semiformal coverage on the more encompassing, larger design partition. In some cases, after (or in parallel to) a higher-level semiformal effort, the higher-level design partition has been further pruned to enable the application of formal techniques to obtain proofs in order to ensure the highest coverage overall. This pruning is performed either by choosing a subset of the higher-level design partition to verify in isolation, or by overconstraining and manually abstracting the larger design partition. In such cases, the semiformal efforts have often been very useful in helping to identify the location of design flaws and complexity within the higher-level design partition, in turn yielding insight into which smaller portions of the design should be targeted by pure formal verification.

As another point, the semiformal algorithms in SixthSense and the bounded-model-checking algorithms in RuleBase support only safety properties. If the goal is to prove liveness properties and design size is amenable, RuleBase is the preferred tool. Otherwise, it is often desirable to attempt to cast the unbounded liveness properties as bounded liveness properties that are safety properties and thus amenable to a wider variety of algorithms.

8. Results

In this section we discuss the application of FFV to portions of the pSeries High Performance Switch, the pSeries Switch Network Interface, and the POWER5 microprocessor. We provide details of the portions of these designs that were verified, the tools used for these efforts, and the results of these efforts.

High Performance Switch (HPS)

The pSeries HPS chip is the switching element of HPS System Area Networks; it is the fourth generation of the IBM Scalable Parallel switch chip [25]. The pSeries HPS supports the high-bandwidth, low-latency network needs of a distributed shared memory (DSM) system. **Figure 5** shows a high-level view of the HPS chip.

FFV was executed on Pass2 and Pass3 of the pSeries HPS chip. Formal verification was not started on Pass2 until well after simulation had stabilized. At this point in time, the design was stable and there was working hardware for Pass1 on the test floor. Thus, the formal verification effort targeted only areas that were difficult to verify in a random simulation environment. The main focus was on the verification of arbitration properties within the output port of the switch chip. The output port

is responsible for transmitting data provided to it by other internal device logic. Each output port performs an arbitration among the competing input port requests, central buffer requests, and service requests. The properties verified for the output port included aspects of the arbitration schemes designed to meet performance requirements. These properties are difficult to evaluate in a random simulation setting. General liveness properties were also verified. Additionally, limited FFV work was targeted on the input port, primarily for performance-related functions.

For the HPS Pass3, FFV support was applied late in the verification cycle to assist the simulation team in validating error injection functionality. The FFV environment was able to drive specified error conditions and determine that the correct local-level error detection and recovery action had been taken. The error injection testing relied heavily on the newly delivered SAT engine in the RuleBase tool.

Table 1 summarizes the work on the HPS. Even though the FFV effort was limited in duration and scope, it uncovered 37 design defects, 10% of the overall number of design problems found throughout the Pass2/Pass3 verification effort.

Switch Network Interface (SNI)

The IBM Scalable Parallel (SP*) Switch2 Adapter, the communication adapter for use with POWER3 systems, was the first project within the SP verification group to apply FFV on a large scale. During the verification effort, FFV located approximately 200 design defects using RuleBase out of a total of 771 defects. Building on its success in the SP Switch2 Adapter effort, the FV team engaged the design and simulation team at the start of the verification process for the follow-on to that design, the pSeries Switch Interface (SI) chip, which is an internal component of the SNI. The SI chip provides message-passing capabilities for eServer p4 systems across a GX bus. The design was modified to provide this capability to eServer p5 systems across a GX+ bus. The function within the SI chip targeted for FFV coverage remained largely unchanged between the two designs.

In contrast to the SP Switch2 Adapter, a key portion of the SI chip is the microcode-driven Inter-Partition Communication (IPC) sequence engine. A high-level diagram of the SI chip is shown in **Figure 6**. The IPC was the primary area of focus for the FV team because it was deemed prone to corner-case bugs. In addition to the IPC, the Transport Macro (FTM) and the Media Access Controller (MAC) arbiter were identified for formal coverage. The FTM is responsible for ensuring end-to-end in-order delivery of data. The MAC is responsible for transmission of packets across multiple virtual lanes. Of an initial eleven FV candidates, seven were chosen for formal analysis. The decision was based on design nature, simulation coverage, and FV resources. **Table 2** lists the partitions verified using FFV. RuleBase was used for all blocks on the SI chip. The Discovery engine was deployed in all rules, and in the majority of rules, complete proofs were obtained. Some restrictions were required in order to control design size to enable proofs. For example, in the MAC arbiter verification, only two of eight virtual lanes were used. The SAT engine proved to be very useful in exposing design defects for rules that were challenging for the Discovery engine. In the largest of the partitions verified, the IPC packet mover, complete proofs were

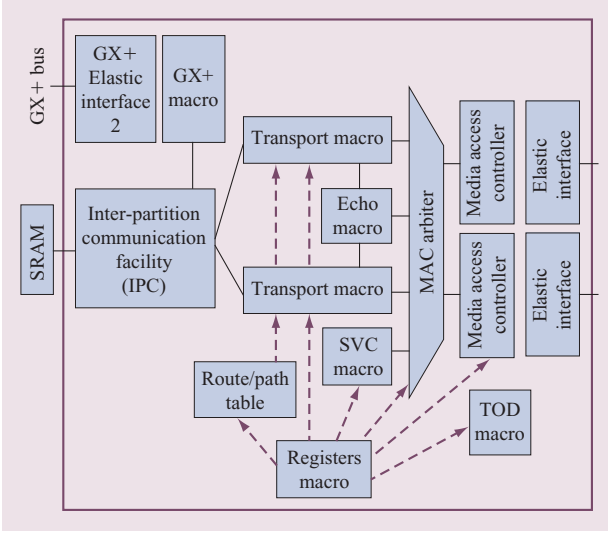

Figure 6

High-level diagram of SNI chip.

Table 1 FFV summary for the HPS chip (RuleBase was used in all cases).

Block name	*Initial model size* (no. of registers)	*No. of inputs*	*Average size of reduced model* (no. of registers)	*No. of rules written*	*No. of completed proofs*	*Engines used*	*Maximum depth of logic*
Output port	2,071	506	70–320	80	62	Discovery, SAT	~3,500
Input port	2,514	549	180–330	34	27	Discovery, SAT	~2,200

Table 2 FV summary for the SI chip (RuleBase was used in all cases).

Block name	*Initial model size* (no. of registers)	*No. of inputs*	*Average size of reduced model* (no. of registers)	*No. of rules written*	*No. of completed proofs*	*Engines used*	*Maximum depth of logic*
MAC arbiter	4,000	1,063	~200	14	14	Discovery, SAT	25
FTM receiver	1,353	1,085	160–200	37	37	Discovery	214
IPC packet mover	8,002	730	200–460	126	106	Discovery, SAT, Unfolding, Beelzebub	788
IPC data mover	700–6,300	175–1,148	50–210	298	268	Discovery, SAT	496
IPC packet mover interface	1,520	347	50–160	32	32	Discovery	42
IPC array interface (arbiter)	39	20	16–33	18	18	Discovery	344
IPC sequencer	1,730–3,269	197–434	70–185	51	36	Discovery	124

not obtained for 20 rules. Two additional engines, Unfolding and Beelzebub, were deployed in these cases. The Unfolding engine was able to uncover bugs that were not exposed by any other engine.

The FFV effort discovered a total of 95 design defects out of a total of 809 defects. Of the 95 defects exposed in FFV, 5% to 10% were characterized as corner-case bugs, which were unlikely to be exposed through random simulation.

Microprocessor

There were two types of formal verification applications on the processor core: proactive work and re-creation of test-floor failures. The goal of the proactive work was to expose design flaws before the design was fabricated. Re-creation of a test-floor failure was undertaken with FFV if a bug exposed on the test floor in the fabricated hardware could be narrowed to a set of design components, and it was deemed difficult for traditional simulation to re-create the test-floor fail condition.

Proactive work

The components of the POWER5 core that were proactively verified with FFV were chosen primarily because of the amount of new and complex functionality added to the component since the prior design generation. One significant cause of such functionality modification was the addition of simultaneous multithreading (SMT) to the POWER5 core.

Applying formal verification early in the design cycle can help to quickly stabilize the design. Design flaws found early in the design cycle permit fixes that can be more robust and extensive because of the amount of time available to perform verification and synthesis on the modified design. An example of such an application was the FFV effort on the Branch Instruction Queue (BIQ). This logic was chosen because of design changes for SMT and the high bug count of its counterpart on POWER4. Formal verification began while the component evolved from the POWER4 implementation to support SMT. This allowed verification of the modifications as they were entered, enabling design defects to be identified immediately and fixed. Formal verification of the BIQ uncovered more than 20 defects, many of which were quite complex in nature.

Later in the design cycle, formal verification is useful to uncover the complex design flaws that simulation has not exposed. As an example, the Instruction Prefetch Buffer (IFPF) was formally verified later in the design cycle. This logic had been problematic in POWER4 and was changed for POWER5. In the POWER4 timeframe, RuleBase was used on the IFPF. While numerous design defects were exposed in that effort, this logic posed size problems for the formal algorithms and had to be significantly constrained. For example, its buffer size was cut in half to reduce complexity for the formal algorithms. For POWER5, SixthSense was available in time for the formal verification effort. This allowed semiformal verification to be applied to the logic without any constraints, exposing six bugs. The majority of these design flaws were very complex in nature and required the unconstrained environment. For example, some of the bugs required more than half of the buffer to be active. One showed a possible input stimulus that the simulation environment was not exercising; the simulation environment was subsequently augmented to exercise this behavior.

With the advent of POWER5, the use of SixthSense to check equivalence between two design points was realized. During the design cycle, the condition register

file was redesigned to improve timing. The design change had to preserve sequential input/output equivalence with respect to its designated initial states. Since the redesign entailed a change in the sequential implementation of the design (i.e., the two versions of the design had different state points), Verity [21] could not be readily used to verify this equivalence. SixthSense was therefore used for this purpose, composing the two versions of the design, correlating their inputs, and adding assertions to check that the corresponding output pins evaluated to the same sequence of values. This effort uncovered three design flaws in the new design. Once the design flaws were fixed, the SixthSense reduction and transformation engines efficiently proved that the two designs were equivalent under all possible input stimulus.

Re-creation of test-floor failures

Re-creation of a test-floor failure is the process of identifying the cause of a logic problem encountered in a fabricated chip. This process has become a powerful use of formal verification, because any design flaw that has reached silicon has already evaded a great deal of simulation. For this reason, formal verification is often desired to increase the confidence in any fixes. Additionally, sometimes the information obtained on the test floor provides only a vague idea of the true location of the problem, and a complete trace of the fail is desired to facilitate a fix. Test-floor data is used to define directed simulation cases to attempt to re-create the failure. For difficult corner-case bugs, however, even directed simulation may be inadequate to recreate it. In contrast, the coverage achievable through formal verification is often powerful enough to obtain a complete trace exposing the design flaw. Sometimes formal verification can illustrate multiple scenarios of a defect, ensuring that the resulting fix is robust and covers all failing scenarios instead of merely the first encountered one. In some cases, the re-creation effort evolves into a proactive bug-finding effort for the next fabrication of the design.

An example of the power of FFV in the re-creation of a test-floor failure was demonstrated in the Instruction Effective to Real Address Translation (ERAT) logic. Formal verification was engaged on this logic after an address translation error was encountered on the test floor. SixthSense was used for this effort because of the size of the desired model, which included the Instruction ERAT logic as well as several neighboring control blocks. After coding the environment and assertion to target the design flaw, an unknown bug was identified in addition to re-creating the known problem. This effort then evolved into a proactive type of verification to check additional functionality prior to the next refabrication. During this work, several other design defects were discovered, all present in the prior fabricated hardware. All of these bugs were very complex corner cases, as is evident by the fact that they had slipped into silicon. Some of these proved too difficult to recreate with directed simulation efforts or in hardware, even given the data from the formal traces.

Microprocessor and memory subsystem work summary

In **Table 3** we provide a summary of the processor design components to which we applied FFV. The Branch Instruction Queue (BIQ) maintains information about branches in case of branch mispredictions. The Data ERAT LRU encodes a least-recently-used algorithm for the Data ERAT. The Segment Lookaside Buffer (SLB) control logic provides arbitration for address translator logic. The Fabric Bus Controller (FBC) logic, as well as the Sidecar logic, comprise data- and control-intensive logic responsible for routing data to and from various internal interfaces and system buses. The FBC Dataflow and MCM model leveraged the power of semiformal verification to compose some of the smaller components into larger partitions, against which more encompassing architectural properties were checked. The IFPF is the instruction prefetch logic. The Completion Subunit tracks the completion of instructions. Both the Data and Instruction ERAT logic were formally verified. The Condition Register (CR) Mapper is a register-renaming unit for the condition registers. The Instruction Fetch Address Register (IFAR) implements program-counter functionality. The SCMD is a memory controller. Finally, the ERAT Miss Queue (EMQ) is a queue for address translations. Overall, a total of 41 design flaws were proactively exposed by these efforts on the microprocessor core and memory subsystem.

9. Summary

Formal verification has matured from an academic interest to a verification method applicable to industrial designs, and the successful deployment of FFV technologies on the IBM pSeries designs reflects this growing maturity. The improvements in formal tools have allowed larger partitions of logic to benefit from formal coverage. In the POWER4 timeframe, a common challenge on the processor core was to simply identify a meaningful partition of logic that was within the size limitations of FFV. That is, when sizing the design for FFV, partitions within the size limitation of the current tools were often deemed "too small" to capture interesting architectural or microarchitectural properties. Larger partitions often required many over-constraints on the design (e.g., disabling or restricting certain behavior of the design) such that it lessened the confidence in the overall coverage attained even when a proof was completed. Thus, while a significant number of high-quality bugs were discovered using FFV during

Table 3 FFV summary for the POWER5 microprocessor core and memory subsystem.

Block name	*Initial model size* (no. of registers after phase abstraction)	*Average size of reduced model* (no. of registers)	*No. of inputs*	*No. of rules written*	*No. of completed proofs*	*Tool used*	*Maximum depth of logic*
BIQ	5,785	98–124	529	63	63	RuleBase	27
Data ERAT LRU	1,012	40–42	63	2	2	RuleBase	5
SLB Control	431	124–125	80	8	8	RuleBase	54
FBC NCU Controller	407	31–47	246	8	8	RuleBase	14
FBC GX Controller	119	34	67	9	9	RuleBase	18
FBC L2 Controller	271	35–60	97	15	15	RuleBase	11
FBC L3 Controller	268	46–75	120	6	6	RuleBase	11
Sidecar	323	117	28	2	2	RuleBase	72
Sidecar + Arbiter	532	221	28	2	2	SixthSense	76
FBC data flow	3,954	946	5,394	4	4	SixthSense	142
FBC MCM Model	3,800	823	40	4	0	SixthSense	unknown
IFPF	11,220	966	464	19	0	SixthSense	unknown
Completion Subunit	17,622	3,027	461	7	0	SixthSense	unknown
Data ERAT	45,637	19,913	6,874	4	0	SixthSense	unknown
Instruction ERAT	9,512	2,072–2,205	1,456	18	0	SixthSense	unknown
CR Mapper	17,355	2,371	604	6	0	SixthSense	unknown
IFAR	3,533	182	763	2	0	RuleBase and SixthSense	unknown
CR Register File	1,398	1,264	84	21	21	SixthSense	unknown
SCMD	2,259	764	105	2	0	SixthSense	unknown
EMQ	1,892	1,086	217	3	0	SixthSense	unknown

the POWER4 core verification effort, the overall formal coverage attained from an architectural or microarchitectural point of view was rather limited. Most design defects tended to reflect violations of lower-level block assertions. Additionally, the formal work done on the POWER4 processor core tended to operate at the level of smaller blocks, whose interfaces were often complicated, not well documented (aside from the FFV effort), and prone to frequent changes. These factors led to an overall increase in the amount of work necessary to obtain formal results. With the improvement in the formal toolset, specifically the introduction of semiformal verification, higher-level properties became candidates for formal coverage in the POWER5 timeframe.

Across the High Performance Switch, the Switch Interface Chip and the POWER5 processor, FFV was used to discover complex design flaws, of which a fair number were characterized as unlikely to be detected by simulation. As FFV has proven its value over the course of POWER4 and POWER5, we expect continued investment in tool development and application on IBM designs. Close collaboration with the simulation and design teams will remain essential as FFV is used to augment simulation coverage, targeting the areas deemed most complex for the traditional simulation environment. Generally, if simulation misses a logic bug, it is because the random environment could not create the conditions necessary to expose it. If FFV misses detecting a logic flaw, the most likely reason is that a rule was not written that would expose it. In future projects, means of better leveraging verification resources between simulation and FFV will be explored, e.g., through reuse of specifications and by increased exploitation of the growing capacity of the formal toolsets to improve the coverage of a larger set of verification tasks. As the formal tools continue to improve and evolve, continued deployment of these technologies in the ongoing effort of improving our hardware designs is expected.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. T. Kropf, *Introduction to Formal Hardware Verification*, Springer, Berlin, Germany, 1999.
2. P. Camurati and P. Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," *IEEE Computer* **21**, No. 7, 8–19 (1988).
3. P. Bjesse, T. Leonard, and A. Mokkedem, "Finding Bugs in an Alpha Microprocessor Using Satisfiability Solver," *Proceedings of the 13th International Conference on Computer-Aided Verification*, Paris, July 2001; *Lecture Notes in Computer Science* **2102**, 454–464 (2001).
4. R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical Formal Verification in Microprocessor Design," *IEEE Design & Test of Computers* **18**, 16–25 (July–August 2001).
5. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, "Benefits of Bounded Model Checking at an Industrial Setting," *Proceedings of the 13th International Conference on Computer-Aided Verification*, Paris, July 2001; *Lecture Notes in Computer Science* **2102**, 275–292 (2001).
6. I. Beer, S. Ben-David, C. Eisner, and A. Landver, "RuleBase: An Industry-Oriented Formal Verification Tool," *Proceedings of the 33rd Design Automation Conference*, June 1996; *Lecture Notes in Computer Science* **1254**, 480–483 (1997).
7. 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 the POWER4 Multiprocessor Systems," *IBM J. Res. & Dev.* **46**, No. 1, 53–76 (January 2002).
8. D. W. Victor, J. M. Ludden, R. D. Peterson, B. S. Nelson, W. K. Sharp, J. K. Hsu, B.-L. Chu, M. L. Behm, R. M. Gott, A. D. Romonosky, and S. R. Farago, "Functional Verification of the POWER5 Microprocessor and POWER5 Multiprocessor Systems," *IBM J. Res. & Dev.* **49**, No. 4/5, 541–553 (2005, this issue).
9. International Business Machines Corporation, "eServer Cluster 1600: pSeries High Performance Switch Planning, Installation, and Service," Publication No. GA22-7951-02, September 2004.
10. R. Kalla, B. Sinharoy, and J. Tendler, "IBM POWER5 Chip: A Dual Core Multithreaded Processor," *IEEE Micro* **24**, 40–47 (March–April 2004).
11. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz, "An Abstraction Algorithm for the Verification of Level-Sensitive Latch-Based Netlists," *J. Formal Meth. Syst. Design* **23**, No. 1, 39–65 (July 2003).
12. M. K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing Simulation with BDDs and ATPG," *Proceedings of the Design Automation Conference (DAC'99)*, New Orleans, June 1999, pp. 385–390.
13. K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.
14. S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal, "Model Checking at IBM," *J. Formal Meth. Syst. Design* **22**, No. 2, 101–108 (March 2003).
15. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proceedings of the 38th Design Automation Conference*, June 2001, pp. 530–535.
16. R. Tzoref, M. Matusevich, E. Berger, and I. Beer, "An Optimized Symbolic Bounded Model Checking Engine," presented at the CHARME 2003 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, L'Aquila, Italy, October 2003; see *http://www.di.univaq.it/charme2003/*.
17. S. Keidar and Y. Rodeh, "Searching for Counter-Examples Adaptively," presented at the 6th International Workshop on Formal Methods, Dublin City University, Ireland, July 2003; see *http://ewic.bcs.org/conferences/2003/iwfm03/*.
18. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh, "The Temporal Logic Sugar," *Proceedings of the 13th International Conference on Computer-Aided Verification*, Paris, July 2001; *Lecture Notes in Computer Science* **2102**, 363–367 (2001).
19. Web version of the RuleBase User Manual; see *http://www.haifa.il.ibm.com/projects/verification/RB_Homepage*.
20. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," presented at the Conference on Formal Methods in Computer-Aided Design (FMCAD'04), Austin, TX, November 2004.
21. A. Kuehlmann, A. Srinivasan, and D. LaPotin, "Verity—A Formal Verification Program for Custom CMOS Circuits," *IBM J. Res. & Dev.* **39**, No. 1/2, 149–165 (January/March 1995).
22. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," *IEEE Trans. Computer-Aided Design* **21**, No. 12, 1377–1394 (December 2002).
23. A. Kuehlmann and J. Baumgartner, "Transformation-Based Verification Using Generalized Retiming," *Proceedings of the 13th International Conference on Computer-Aided Verification*, Paris, July 2001; *Lecture Notes in Computer Science* **2102**, 104–117 (2001).
24. H.-W. Anderson, H. Kriese, W. Roesner, and K.-D. Schubert, "Configurable System Simulation Model Build Comprising Packaging Design Data," *IBM J. Res. & Dev.* **48**, No. 3/4, 367–378 (May/July 2004).
25. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, "The SP2 High-Performance Switch," *IBM Syst. J.* **34**, No. 2, 185–204 (1995).

Received November 11, 2004; accepted for publication March 29, 2005; Internet publication August 11, 2005

Rebecca M. Gott *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (gott@us.ibm.com).*
Dr. Gott joined IBM in 1999; she is currently an Advisory Engineer with the IBM Systems and Technology Group. For the last four years, she has worked in the area of functional formal verification, and for the last three years has served as the team leader for functional formal verification within the IBM Advanced Processor Development organization. Most recently she has worked on the formal verification of the POWER6* microprocessor. In 1995, she received her Ph.D. degree in electrical engineering from Tulane University. Following her graduation, she served on active duty in the U.S. Air Force at Holloman AFB, New Mexico, working in the area of GPS development and test. Dr. Gott has recently taken a new position at IBM as the simulation leader for the zSeries* processor subsystem.

Jason R. Baumgartner *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (baumgarj@us.ibm.com).*
Dr. Baumgartner is a Senior Engineer and Master Inventor with the IBM Systems and Technology Group. He received his Ph.D. degree in electrical and computer engineering from the University of Texas at Austin in 2002. His research interests have been fueled by years of functional formal verification applications at IBM. He has pioneered numerous automatic abstraction techniques to yield exponential speedups to formal analysis, which are the subject of numerous papers, patents, and his doctoral dissertation. Dr. Baumgartner is currently the team leader for the IBM internal semiformal verification toolset SixthSense. He recently received an IBM Excellence Award and an IBM Outstanding Innovation Award for his formal verification application, research, and development work on POWER4 and POWER5.

Paul Roessler *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (roessler@us.ibm.com).*
Mr. Roessler joined IBM at Austin in 2001. He has spent his career working in the area of applied functional formal verification on microprocessor designs. He received a B.S. degree in electrical engineering from New Mexico State University in 1998. Mr. Roessler recently received an IBM Outstanding Technical Achievement Award for his POWER5 formal verification applications. He is currently working on the functional verification of the POWER6 microprocessor.

Soon I. Joe *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (soonjoe@us.ibm.com).*
Mr. Joe received a Master of Science degree in electrical engineering from Drexel University in 1981, joining IBM that same year at Charlotte, North Carolina. He is currently an Advisory Engineer with the IBM Systems and Technology Group. He has held a variety of test and logic design positions throughout his career. Since 1999 he has worked in the area of functional formal verification. Mr. Joe is currently working on the functional verification of the POWER6 microprocessor memory subsystem.