Enabling Large-Scale Pervasive Logic Verification through Multi-Algorithmic Formal Reasoning

Tilman Glökler[1] Jason Baumgartner[2] Devi Shanmugam[2] Rick Seigler[2]
Gary Van Huben[2] Barinjato Ramanandray[1] Hari Mony[2] Paul Roessler[2]

[1]IBM Deutschland Entwicklung GmbH [2]IBM Systems & Technology Group

Abstract*— *Pervasive Logic is a broad term applied to the variety of logic present in hardware designs, yet not a part of their primary functionality. Examples of pervasive logic include initialization and self-test logic. Because pervasive logic is intertwined with the functionality of chips, the verification of such logic tends to require very deep sequential analysis of very large slices of the design. For this reason, pervasive logic verification has hitherto been a task for which formal algorithms were not considered applicable.

In this paper, we discuss several pervasive logic verification tasks for which we have found the proper combination of algorithms to enable formal analysis. We describe the nature of these verification tasks, and the testbenches used in the verification process. We furthermore discuss the types of algorithms needed to solve these verification tasks, and the type of tuning we performed on these algorithms to enable this analysis.

I. Introduction

High-performance digital designs such as the Cell Processor [1] and Pentium [2] generally have extremely aggressive clock frequency goals, which requires manual optimization of the logic, circuits, and even the layout of critical portions of the design. Such manual optimization amplifies the risk of functional design errors, which must be identified and rectified as early as possible in the design flow to avoid costly design and manufacturing iterations.

Apart from the primary functionality of a design such as *arithmetic* or *instruction decode* logic, virtually all modern designs include *Pervasive Logic* (PL) [3]. The PL includes logic for controlling the power-on-reset initialization sequence, for security features to boot only trusted software, and for enabling manufacturing test capabilities such as built-in self-test for logic and memory arrays. It also includes debug functionality for trace logic analysis and to enable access to internal latches and arrays in the chip.

The PL of a design must be customized to the needs of the specific design microarchitecture and circuit technology, thus, it is often entirely manually designed. Furthermore, mainly for timing reasons, this custom PL must often be manually integrated into the functional logic itself, and the two are tightly intertwined. For example, the *scan chains*, which serially connect many of the latches in the chip for initialization, tracing, and test purposes, are often manually connected within the design HDL itself. This greatly contributes to the risk of implementation errors within that pervasive logic or even the functional logic itself. While the goal of functional verification is to validate that the design correctly implements its specification despite the intertwined PL, the goal of *Pervasive Verification* (PV) is to further guarantee that the PL works as intended. In many ways, the correctness of the PL is *more* critical than the correctness of the functional logic, since an error in the latter may well render a costly fabrication of a chip entirely unusable or untestable, whereas a functional logic error at least enables the analysis of other aspects of the chip and may often have a software or hardware workaround.

A. Pervasive Verification Tasks

Most pervasive verification tasks can be subdivided into validating the following categories of PL.

a) POR, Security, and eFuses: Power-on reset (POR) refers to the procedure of initializing or *booting* the chip after enabling its voltage supply, and is responsible for ensuring that the chip is brought to a consistent initial state to ensure proper functional behavior when control is handed over to software execution. During POR, almost all other pervasive functionalities are needed such as initialization of latches and arrays in the proper order, sensing of the security keys in the electric fuses (eFuses) [4], processing security information to ensure that only trusted software can be run on the system [5], bringing up the physical Input / Output interface, enabling runtime error analysis in debug mode, etc.

b) External Debug Interfaces: External debug interfaces such as JTAG [6], SPI [7], and I^2C [8] are used to debug logic errors and analyze manufacturing problems in a chip. These interfaces have access to debug registers which control various functions in the chip such as clocking and Input / Output setup. These serial interfaces are also used to access the scan chains and, thus, enable access to nearly all latches and arrays in the chip. Special features for supporting the reliability, availability and serviceability of the chip allow handling of recoverable or unrecoverable errors (e.g., uncorrectable memory errors), and provide an interface to system software. Using these mechanisms, error recovery can be performed on-the-fly by system software.

c) Debug and Built-In Self-Test Logic: The scan chains in the chip are used for various purposes including initialization during POR, reading latch values onto off-chip interfaces, setting the chip into specific configuration modes, and built-in self-test.

The trace bus and trace logic analyzer functionality is used to monitor thousands of chip-internal signals in real-time, non-intrusively, while the chip is functionally running. This bus

is configurable either by a scan operation or other dedicated registers, and can select among various internal signals to be analyzed by an on-chip evaluation circuit. This functionality allows one to observe transient events in a running chip, in contrast to scan-based analysis, which requires the chip to temporarily suspend functional operation while scanning. The trace logic analyzer supports many functionalities similar to a desktop logic analyzer such as triggering on configurable conditions in real-time, tracing of signals before or after a trigger has occurred, etc. The results of the trace memory can be used for performance monitoring of the processor core or in order to obtain waveforms of internal signals for debugging purposes in the bring-up lab.

The array built-in self-test (ABIST) functionality is used to detect fabrication defects in all on-chip memories (RAM and ROM). The so-called *ABIST engines* apply parallel read and write pattern tests to the memories to detect such faults. If a fault is detected, the ABIST engine has the capability to identify and report the fault, along with information on how to "repair" this fault using redundant bit or word-lines, or other more specific redundancy schemes. The ability to access internal array cells for debugging is also enabled by ABIST.

Logic built-in self-test (LBIST) [9] is used to detect manufacturing faults in the logic. The LBIST controller has a Pseudo-Random Pattern Generator [10] which generates test-patterns. LBIST iteratively uses two different phases: a scan phase which initializes the LBIST scan chain stumps with pseudo-random or random-appearing yet deterministic data, and a functional phase, which clocks the latches for several time-steps. The next LBIST iteration utilizes the scan chain stumps to scan values into multiple-input signature registers (MISRs). In case of a chip with manufacturing faults, the MISR signature will most likely differ from the expected value burned into an unfaulty chip. Verification must ensure that these LBIST phases work as expected; i.e., that all scannable registers can be initialized correctly, that the functional updates do not propagate an uninitialized state to the non-scannable latches, and that the MISR patterns are fully controllable and match in different LBIST modes.

d) Fencing Logic: Fencing logic at chip or partition boundaries ensures that a certain chip or partition is isolated from the surrounding logic while being reset, reconfigured or while running LBIST. The purpose of fencing logic in such a scenario is to provide safe and deterministic input values to the chip / partition. For LBIST, deterministic input values are essential in order to obtain reproducible signatures in the MISRs. Reconfiguration of a partition also requires safe input values in order to guarantee that the final state after reconfiguration is as expected. Fencing logic can be as simple as one logic gate connected to every primary input and a *fence enable* signal that collectively force safe values at internal signals. In other cases, registers are used to implement the fencing logic, which must be initialized to safe values and then forced to hold their state while the fence is enabled.

e) Time Reference: Time reference is an important functionality for systems such as servers with multiple CPUs distributed across different locations. Its purpose is the synchronization of the time-of-day clocks to ensure a consistent time-stamp data across multiple servers and operating systems, e.g., to enable synchronized database accesses.

B. Pervasive Logic Verification Challenges

One challenge of PV is that the design specification is extremely project-specific and the logic is often designed anew for each chip. This results in a specification challenge: the specification cannot be reused to any extent from design generation to generation, unlike many other architectural components of the chip. Another main issue of PV is due to design complexity. In contrast to partition or unit verification, many features of pervasive logic cannot be verified in an isolated block or unit [11]. As the pervasive logic is intertwined with the functional logic, PV has to work with very large components of the design – in cases, even full chip-level models. The cones of influence of the PV properties often span hundreds of thousands of state elements due to aspects such as requiring the use of long serial scan chains [12] and interfacing with very large memory arrays.

The size of the design components required for many PV tasks has historically precluded their receiving any substantial focus from the formal verification community. Furthermore, the sequential depth of many PV tasks – e.g., to serially scan data through possibly hundreds of thousands of latches in a design component – further complicates formal reasoning, e.g., to preclude efficient inductive analysis. Due to these challenges, simulation and hardware emulation have traditionally been used to validate many pervasive features, along with static analysis tools that validate aspects such as scan chain connectivity. While useful for falsification, such approaches are generally incomplete and cannot guarantee the absence of design flaws.

II. Functional Verification Testbenches

The verification paradigm we adopt in this paper is that of a *testbench*, wherein one develops a set of property automata or *checkers* to assess the correct behavior of the design, in addition to a *driver* to constrain the input stimuli to which the design may be subjected to avoid spurious failures. Initialization data is also generally provided for the design. For example, one may initially randomize the state elements of the design, and use the driver to walk the design through a reset sequence prior to performing functional verification. Or, for computational efficiency, one may directly restrict the initial states of the design to those guaranteed by such a power-on reset sequence, without requiring each verification run to begin with performing an explicit reset sequence. The verification task thus consists of trying to obtain a counterexample trace from a specified initial state to one which drives a logical *one* onto the output of a property automata (in the composition of the design with its checkers and drivers), or proving that no such counterexample exists.

Given such a testbench, one may deploy a variety of algorithms to attempt to solve the corresponding properties. For falsification, one may wish to utilize random simulation, hardware emulation, or semi-formal analysis. For verification, one may wish to deploy proof techniques such as reachability

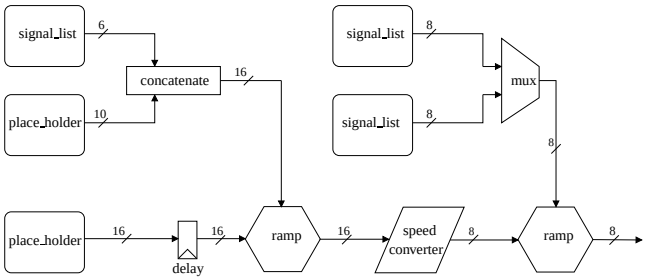

Fig. 1: Example Trace Bus

Primitive Block Type	Description
signal_list	References design signals as data inputs to the trace bus.
mux	Drives the *source* block selected by a user-defined function of a *selector* block.
ramp	OR's two blocks by a user-defined function of an *enable* block.
speed_converter	This block specifies the transfer of data (another block) across asynchronous clocking boundaries with user-defined behavior.
concatenate	Concatenates other primitive blocks to form a *wider* block.
extract	References only a *subset* of another primitive block.
place_holder	Like **signal_list**; a place-holder for incomplete descriptions.

TABLE I: Primitive blocks used for the *tracedef* language

analysis [13] or induction [14]. To help compensate for the generally exponential resource dependency between the size of the testbench and the resources needed for falsification or verification, various transformation and abstraction techniques have been proposed to automatically reduce this size.

In these experiments, we use the IBM internal verification toolset *SixthSense* [15]. The set of verification and transformation engines we report in our results include the following.

- **COM**: a combinational optimization engine, which attempts to merge functionally equivalent gates and rewrite logic cones to reduce their overall size [16], [17].
- **EQV**: a sequential redundancy removal engine, using a more general set of algorithms to solve a *van Eijk*-style induction problem to identify and merge gates which are sequentially redundant [18], [19], [20].
- **RET**: a min-area retiming engine, which attempts to reduce the number of registers in the netlist by shifting them across combinational gates [21].
- **CUT**: a reparameterization engine, which replaces the fanin-side of a *cut* of the netlist graph with a trace-equivalent, yet simpler, piece of logic [22], [23].
- **LOC**: a localization engine, which isolates a cut of the netlist local to the properties by replacing internal gates by primary inputs [23]. This transformation is sound but incomplete – proofs of correctness on the localized design are valid for the unlocalized design, but counterexamples may be spurious. To help guide the cut-selection process, the engine uses a light-weight SAT-based refinement scheme to include only that logic which is deemed necessary [24].
- **RCH**: a BDD-based reachability engine [25].
- **IND**: a SAT-based induction engine which uses unique-state constraints [26].
- **BMC**: a SAT-based bounded model checking engine [16].

Our system uses a high-performance circuit-based SAT solver with intertwined BDD-based analysis, BDD- and SAT-sweeping for redundancy removal, and structural rewriting algorithms, similar to [16].

III. Trace Bus Verification

The so-called trace and debug bus is a global on-chip bus that enables observation of internal signals for debugging and performance monitoring. This bus is configurable using hundreds of registers, and routes subsets (e.g., 128-bit slices) of many thousands of monitorable points to an on-chip logic analyzer unit. The trace bus represents a major challenge for verification, because the available English specification is often ambiguous, and the straightforward approach to use directed testcases would be extremely time-consuming, lossy in coverage, and error-prone.

We addressed these issues by defining a high-level specification language, called *tracedef*, that allows a simple and concise description of such a bus. This *tracedef* specification is used both for documentation (in place of an English specification) as well as input for automated formal testbench creation. As with the use of any concise formal specification language, the use of this language minimizes the risk of specification errors. Instead of using a more standard language such as PSL [27], we chose to utilize our own language, which provided a more concise description of the trace bus using primitive blocks that closely correspond to the typical logical elements comprising such a bus as described in Table I. These elements also have associated *delays* to reflect those present in the actual design. The tracedef language describes the trace bus as a tree structure. The leaves of the tree are the **signal_list** and the **place_holder** blocks, which model the inputs to the trace bus. The other primitive blocks in Table I represent the tree nodes. Each reference from one block to another results in an edge in the tree, and models a connection in the design. The primitive block representing the root node of the tree represents the output of the bus. Figure 1 illustrates an example structure of such a trace bus.

We developed an automated testbench creation process, which automatically creates drivers and checkers from a *tracedef* specification. The checker is a reference model of the trace bus using library elements for each of the *tracedef* primitives against which the actual trace bus implementation, intertwined with the functional design logic, is checked for equivalence. The driver injects nondeterministic cutpoints at the **signal_list** and **place_holder** elements, though ensures that only valid trace bus configurations are taken into account for this equivalence check. For instance, a multiplexer implementation might use NAND-NOR logic requiring the control register to be one-hot in order to obtain a valid multiplexer behavior, whereas other implementations may support arbitrary selector values. Our testbench creation process extracts all valid selector values from the *tracedef* specification, and constrains the testbench driver to allow only those valid settings. Additional configuration data, which we term *traceconf*, are used to specify constraints for primary inputs such as clock frequencies, necessary reset signal setup, constant enable signals, etc.

The design we verified posed an additional challenge since

Metric	Initial	COM	LOC[1]	COM	EQV	Resources
Inputs	33441	21492	11	11	0	792s 624MB
Gates	924723	797710	596	493	0	
Registers	142072	125520	193	193	0	
Properties	128	128	1[1]	1	0	

TABLE II: Memory Flow Controller Results. [1]Localization performs a case split, solving each property independently; the largest localized cone is reported. A column with 0 properties reflects that the corresponding engine solved the problem. Resources reported are cumulative for all 128 properties.

Metric	Initial	COM	LOC[1]	CUT	RET	COM	Resources
Inputs	4188	2878	173	151	238	0	14508s 412MB
Gates	271270	144559	857	1193	1292	0	
Registers	83880	33322	370	370	109	0	
Properties	1381	700	1[1]	1	1	0	

TABLE III: Load Store Unit Results. [1]Localization performs a case split; the largest localized cone is reported. Resources reported are cumulative for all properties.

it uses multiple latch and register types, which are sensitive either to the rising or the falling edge, or various levels, of the clock. This complicated the equivalence check using our simple automatically-generated reference model, which consisted of just one register type for simplicity. We solved this problem by constraining the driver cutpoints at the trace bus data inputs to arbitrary nondeterministic values that cannot toggle more frequently than once per clock period. This is a conservative constraint since in the actual design, these signals are each driven by one type of state element hence cannot toggle more frequently than once per clock period. With this constraint, we eliminated spurious mismatches due to our reference model sampling the cutpoints at a different clock phase than the actual design.

A. Verification Results

One of the larger design slices that we specified and tested using the described methodology is that of a Memory Flow Controller subsystem. This subsystem includes an L2 cache which is inclusive of L1 cache data, and non-cacheable units, which handle cache-inhibited requests from the processor core. There is a controller associated with the cache which handles various operations including the cacheable load and store requests from the core and the memory management unit.

For the memory flow controller there are 128 properties, which are one-bit reference-model checks corresponding to each output of the 128-bit trace bus. Because the trace buses do not route data from or through memory arrays, to simplify the formal verification task we black-boxed all arrays, replacing their output ports by nondeterministic cutpoints. Table II provides the results of solving these properties with a multi-algorithm flow. We report problem size in terms of the number of nondeterministic input variables, combinational gates (in terms of synthesis down to 2-input AND gates), registers, and properties. In addition to a relatively large netlist size, the memory flow controller spanned nearly 16,000 lines of design VHDL. If all of the library files used by the memory flow controller are included, there are more than 300,000 lines of VHDL. As discussed in Section I, the trace buses were intertwined with the design logic in the VHDL, complicating the overall verification task.

Nonetheless, as illustrated by Table II, the ability to leverage the proper algorithm flow enabled us to solve these problems efficiently in approximately 13 minutes. All experiments reported in this paper were run using a single processor of a 16-way 1.9GHz POWER4 system. The optimal solution first employed low-cost combinational optimization across all properties, simplifying subsequent localization analysis. After a dramatic reduction through localization, each subsequent localized property was solved efficiently using combinational optimization followed by van Eijk-style induction. Though each property represented a bit-slice of the trace bus, isomorphisms among the properties were broken at numerous points due to intertwined BIST chains and lack of isomorphisms among the functional logic being sampled by the trace bus.

We also applied this methodology in numerous places in the processor core. For example, we applied the technique to the load-store unit, again black-boxing the larger memory arrays for reduced resources. The results of this verification are provided in Table III. Though somewhat smaller than the memory flow controller, this run took nearly 4 hours to complete. The longer run-times were partially attributed to the larger number of properties, and partially to more complex control logic requiring more post-localization transformations before proofs became feasible. These transformations included reparameterization and min-area retiming. After retiming, the resulting problem became a tautology easily discharged by combinational optimization. In contrast, without localization, retiming was unable to sufficiently simplify the problem to render tautologies. Without these transformations, induction alone was very expensive and could not solve the properties within 48 hours.

Dozens of design flaws were encountered during these efforts. The most basic, as would be expected, were that incorrect signals were propagated through the debug bus with incorrect timing as compared to the specification, typically due to improper multiplexor selector implementations or bad wiring of the bus. Some of the more intricate flaws were due to clock gating controls disabling certain latches when they were needed to route data, or speed conversion logic sampling signals with improper timing.

IV. ABIST Verification

Array Built-In Self-Test (ABIST) logic is used in a chip to identify manufacturing defects such as stuck-at faults or short circuits in a memory array [28]. Such logic consists of an *ABIST engine* connected to one or more arrays. The ABIST engine drives address and control information, along with specific write-data patterns, into the scan latches adjacent to the array. The chosen write-data patterns are carefully selected so as to attempt to cover all possible fault types as efficiently as possible. The ABIST engine then triggers the writing of the scanned data into the array using dedicated communication latches. These latches act as pipeline stages to enable the shared ABIST engine to reside further from the array itself

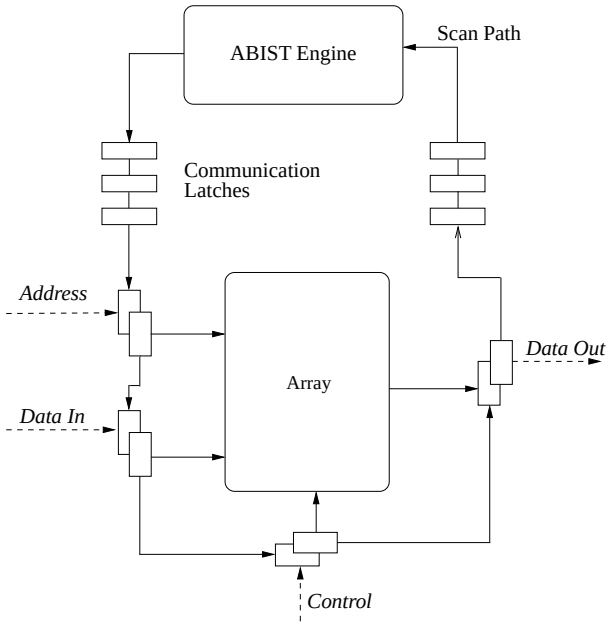

Fig. 2: ABIST Design

than otherwise possible in silicon due to timing requirements and circuit delay.

After sequentially writing all rows and columns of the array, the ABIST engine next reads the written content out of the array in the same order that it was written. This read-out data is then compared against the expected data. Any deviation from the expected data is reported as a fault in the array structure.

Using such a scanned ABIST approach, the ABIST engine cannot test the array faster than the number of clock periods necessary to serially scan all address, control and data bits required to trigger the writes and reads of all necessary patterns and all array cells. In order to alleviate this problem, *shadow* scan latches are added for the address and control path of the array. These latches are interleaved to allow multiple addresses to be applied on successive clock cycles, eliminating the bottleneck of the serial communication channel between the ABIST engine and array. While merely an optimization technique for the ABIST performance, such constructs significantly increase the complexity of the ABIST logic and its verification.

Many modern array implementations provide *repair* capability: if a *repairable* fault is detected, the array may utilize redundant bit or word lines to mask a specific number of repairable faults for subsequent chip operation [29].

Verification of ABIST logic typically consists of artificially seeding errors into the array, and validating that the ABIST engine correctly identifies them, and if applicable, provides information on how to repair them. The main goal of leveraging formal analysis in ABIST verification was to be able to exhaustively test the response of the ABIST engine against the very large set of possible error types. To speed up verification, many ABIST implementations have the ability to program the ABIST engine to cover only specific portions of the arrays: e.g., a smaller set of rows and columns than the large number of rows and columns that comprise a cache. We exploited this capability when performing verification of the ABIST design. Nonetheless, due to the sheer size and sequential depth of these verification runs, attempting any form of exhaustive simulation becomes computationally infeasible. Though these factors also posed great challenges to formal analysis, the availability of the proper set of algorithms ultimately enabled symbolic algorithms to scale to this task.

A. Verification Results

To compensate for the sheer size of the ABIST logic, comprising an entire L2 cache plus the dedicated scan and controller latches, in addition to the ABIST engine itself, our formal testbenches programmed the ABIST engine to operate only upon specific slices of the L2 cache. Without operating on single slices alone, the size of the given testbench was 5,321,918 state elements and 32,143,002 gates. The smallest testbench we could obtain for an array slice, which was still big enough to provide meaningful verification results, contained 246,302 state elements.

The properties we developed for the testbench checked that:

- The state machines of the ABIST engine properly transition from write phase to read phase to compare phase, and finally properly report the end of the testing phase.
- The ABIST engine properly "times" the sending of data to the individual arrays given the pipeline depth of the communication channels between them.
- The ABIST engine properly communicates with the proper arrays.
- Injected errors are properly detected by the ABIST engine.
- The ABIST engine properly indicates whether a repairable number of errors was detected.

Because the goal of the ABIST engine is to properly detect errors, our testbench performed single- and multiple-bit error injection by randomly selecting array cells to "corrupt" by altering the data written by the ABIST engine in transit to the arrays prior to the read and compare phases.

Table IV summarizes our verification results for one slice. Due to the nature of this logic, few transformations were useful to reduce it; isomorphisms were broken by the nature of the ABIST engine and pipeline stages, and there was little redundancy to be exploited. More aggressive transformations such as sequential redundancy removal were time-consuming given the size of the logic, and not very powerful in their reductions. Reachability was clearly infeasible in this domain, and due to the sequential depth of the ABIST engine's sequence, induction also became infeasible. We thus resorted to *bounded* unreachability analysis using BMC. Luckily, since the duration of the ABIST engine process is readily quantifiable, this bounded unreachability approach provided full confidence of correctness of the checked properties.

The behavior of the ABIST engine is largely deterministic; once triggered, it walks through a long execution stream. The primary sources of nondeterminism include the black-boxing done to prune the testbench down to a single slice, and the random selections for error injection. The six properties from

Metric	Initial	COM	**BMC** 440,000	Resources
Inputs	29176	29086	0	1611s 5976MB
Gates	1567812	1394640	0	
Registers	246302	230495	0	
Properties	6	6	0	

TABLE IV: ABIST Engine Results

Metric	Initial	COM	EQV	IND	Resources
Inputs	548878	32362	843	0	211s 748MB
Gates	748426	245309	43978	0	
Registers	73368	23560	5922	0	
Properties	4665	4665	1837	0	

TABLE V: Fencing Logic Results

Table IV include validation that the ABIST engine cycles through its write, read, and compare phases properly, and that errors are properly identified by the ABIST engine. Note that we completed a bounded analysis of 440,000 time-steps for these checks. The "ABIST check done" occurred at time 30,407, giving us confidence that the ABIST engine properly detected all forms of failures. For this reason, we could have concluded that a 30,407-step BMC was adequate to infer correctness - though analysis beyond the critical time-frames was trivial given the nature of our BMC engine.

Our BMC engine uses a structural SAT solver applied in an incremental manner, time-step by time-step. It is highly tuned to unfold only critical signals at critical time-steps, leveraging the simplification performed at earlier time-steps to reduce the size of unfoldings of later time-steps in addition to reusing learned clauses. For ABIST verification, the actual symbolic reasoning about the unfolded instance is not very difficult for the SAT solver; the key was tuning the BMC infrastructure for large designs and very deep unfoldings. For time-frames beyond that which the ABIST engine was operating, structural analysis alone detected that the unfoldings were trivial. It is worth noting that, after tuning our SAT solver in this manner, BMC became several orders of magnitude faster than even random simulation for this large design, since the latter could not be as optimally tuned to operate only on critical portions of the design over time.

Numerous design flaws were exposed during the ABIST testing, including faulty reporting of array errors and incorrect staging of communication channels between the ABIST engine and arrays.

V. Fencing Logic Verification

Fencing logic is typically a small yet essential part of systems with multiple chips or multiple clock domains. The intent of such logic is to prevent spurious logic activity while the system is in the process of being reset, reconfigured or while running LBIST. For example, when an incoming fence signal is active for a particular domain, the internal logic of the receiving domain must be impervious to random transitions that may occur on any number of incoming interface buses or signals [30].

Verification of fencing logic requires demonstrating that all logic associated with a particular fence or set of fences is effectively quiesced during any window of time that the fences are active. For optimality of the design, the fencing logic itself is often kept minimal, risking the exposure that certain input stimulus may erroneously sensitize transitions in the fenced design. Since a single fence line typically serves to protect a multitude of interface signals from interacting with a large amount of downstream logic, simulation alone is often insufficient to expose all possible logic interactions and flaws. This renders simulation-based approaches largely insufficient to yield acceptable coverage, and motivates a formal verification approach.

Due to the straight-forward nature of the verification task, we were able to develop a largely automated formal testbench creation paradigm as follows.

- A property is automatically generated for each register in the fenced design region, checking that the register's value does not alter during the fencing condition.
- All fencing-control inputs are configured to enable fencing.
- The remaining design inputs are categorized from the design specification as being *fenceable* vs. *unfenceable*. Fenceable inputs are precisely those which the fencing logic is required to shelter the design from being sensitized to, hence these inputs are left unconstrained in the testbench. Others may come from adjacent logic blocks which themselves are to be fenced, hence these are driven to arbitrary constant values to avoid rendering spurious failures.

A. Verification Results

We deployed our fencing logic verification methodology on a custom data-flow chip. For our first deployment, before we had a well-tuned sequential redundancy removal engine, we limited our verification to 21 individual units of that chip, each of which was limited to several thousand state elements which were solvable within 10 minutes. Unit-level inputs from other fenced blocks were treated as *fenceable* as per the above methodology, and driven constant; all others were randomized.

Later in the project, a highly-tuned **EQV** engine scalable to larger designs became available. This engine trivialized those unit-level runs to being solvable within a matter of seconds. We thus found that we were able to apply this methodology at the level of the entire chip, though black-boxing the logic arrays since that had no impact on the fenced logic. With this methodology, we were able to leave all chip inputs nondeterministic (aside from those enabling fencing), relying upon the inter-unit connections to effectively propagate the fenced constants throughout the chip. This saved manual effort, since we no longer had to categorize inputs as fenceable vs. non-fenceable. This also eliminated the risk of missed design flaws associated with mis-categorization. Table V provides the results of these experiments.

Six design flaws were identified during this testing, wherein the fencing logic was inadequate to prevent the sampling of design input values into the fenced logic.

VI. External Time Reference (ETR) Verification

ETR denotes a mechanism to keep all processor cores in all nodes of a system synchronized to the same accurate

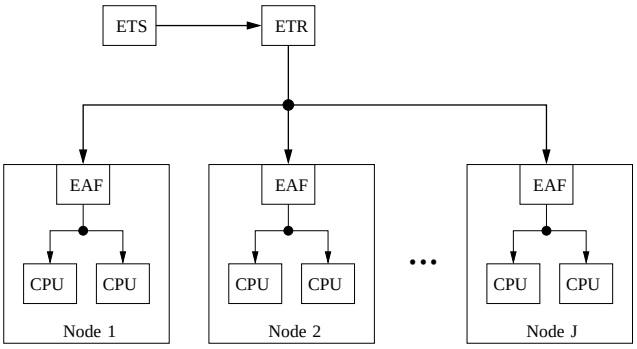

Fig. 3: ETR System

"Time Of Day" [31]. Such a mechanism is very important, for example, in synchronizing database accesses. Each node in a system is connected to the ETR via the ETR Attachment Facility (EAF). The ETR itself is connected to an External Time Source (ETS), usually an atomic clock. Such a system is illustrated in Figure 3.

The ETR generates timing information from the ETS, encodes it and sends it serially in the form of a bit stream to each EAF. For enhanced reliability, this information is redundantly transmitted across two identical channels, and the ETR can dynamically switch between channels if it detects errors. The EAF decodes this bit stream from the selected channel, stores the results and propagates them to the individual CPUs.

Each EAF channel has two main components: a decoder and a store unit. In the decoder, the incoming stream is first sampled at the frequency of the local clock, then passed to an edge-detection unit to extract a bit stream from the samples. The extracted bits are then packed into 16-bit symbols by a deserializer. Afterwards, the decoded symbols are passed to the store unit, which categorizes them as control vs. data words of the following types. The data symbols include Data Byte Symbols (DBS) and Longitudinal Redundancy Check symbols (LRC), which serve as a parity-check for the DBS. The control symbols include the following:

- Idle (IDL), which is used to synchronize the EAF to the incoming stream.
- Data frame start (FST), which is used to flag the start of a data frame containing DBS symbols.
- On-time symbol (OTS), which is used to indicate the end of the stream, and triggers the propagation of the timing information decoded by the EAF channel.

The ETR periodically sends a pattern composed of these symbols, beginning with a sequence of IDL symbols and terminated by one OTS. Each pattern may contain numerous duplicates of the data frame. The rest of the pattern is filled with IDL symbols. Figure 4 illustrates such a pattern.

Before the channel stores and begins processing the timing data, it needs to be synchronized to the incoming ETR stream. This synchronization is achieved by detecting and aligning against a long sequence of IDL symbols. Once synchronization is achieved, the data symbols are stored and validated by comparing the received LRC symbol to a locally-generated value. If the comparison matches, the data is considered to be valid and the subsequent received data frames are ignored,

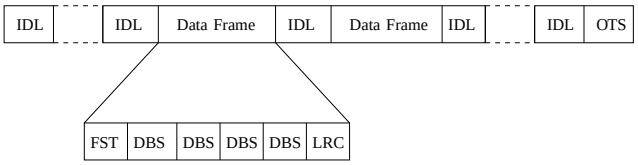

Fig. 4: Example ETR Pattern

since they are only duplicates. If the comparison fails, the process is repeated with the duplicate data frames until valid data is detected. On detection of an OTS symbol, successfully stored data is propagated to the processors. Otherwise, the cause of the failure is reported, such as failure to synchronize, data errors, or missing OTS.

A. Verification Results

The type of properties we verified of the ETR included aspects such as correct symbol decoding, correct synchronization to the incoming signal, proper storage and propagation of data, and proper error detection and handling.

A black-box approach of verifying the overall ETR and EAF logic was quite challenging for formal verification. This system comprised a large number of state elements even after all transformations had been deployed, and was sequentially very deep. A white-box approach was therefore adopted using two testbenches: the first testbench deals with the serial part of the unit, namely the decoding of the bit stream. This testbench is used to validate properties for symbol decoding and synchronisation. The second testbench deals with the parallel part of the unit, namely the storing of the timing information and the switching between the two channels. This testbench is used to validate properties about proper data storage and error handling. The actual design logic was identical across both testbenches; however, the drivers and checkers tied in to the design at different points.

One example property verified of the serial testbench is that every symbol coming from the ETR is correctly decoded. The driver of our testbench was configured to drive a stream of random symbols to the ETR. Because the stream is entirely nondeterministic, we constrained our testbench to validate only the decoding of a single arbitrary symbol. Because the necessary amount of time to decode a symbol was fixed, a BMC approach was adequate to complete this verification task. Nonetheless, the depth of the check was quite large, primarily because the clock frequencies across the ETR and EAF differ by a factor of 96 (hence we used oscillators of periodicity 2 and 2×96 in our driver to clock the two components). The results are depicted in Table VI.

One example property we verified of the parallel testbench is that the detection of the OTS occurs properly. In particular, we built a driver that randomly determines the length of streams it will send from the first data frame to the final OTS (refer to Figure 4). The property verified that if the OTS occurs outside of the allowed tolerance, a missing OTS interrupt is generated. The results are depicted in Table VII. Because the ETR system is architected to be extremely robust, the timeout period for a missing OTS is quite high, hence this

Metric	Initial	COM	EQV	COM	BMC 6,162	Resources
Inputs	2957	18	16	16	0	
Gates	72340	49489	3460	3264	0	843s
Registers	9544	5576	725	725	0	
Properties	3	3	3	3	0	324MB

TABLE VI: ETR Serial Property Results

Metric	Initial	COM	EQV	RET	COM	RCH	Resources
Inputs	319	296	26	38426	26	0	
Gates	73523	49727	798	87196	778	0	78028s
Registers	12922	8720	3332	1725	1724	0	
Properties	10	10	10	10	10	0	8.4GB

TABLE VII: ETR Parallel Property Results

reachability computation required 17,698 image computations for convergence.

VII. Concluding Remarks

Pervasive logic refers to a variety of functionality included in hardware chips orthogonal to their primary architectural functionality, such as logic for initializing the design, for security purposes, and for self-test. Verification of pervasive logic is challenging for a variety of reasons, including the facts that pervasive logic is entirely customized for each chip hence verification and specification reuse tend to be infeasible; the pervasive logic is intertwined with the functional logic of the design at all hierarchies; and most pervasive logic verification tasks require very large design slices (10,000s to millions of state elements) and require very deep temporal analysis. These complexities have historically limited pervasive logic verification to being amenable only to simulation or emulation-based analysis.

In this paper, we discuss how a robust multi-algorithmic formal framework has made a variety of pervasive logic verification tasks feasible and efficient. Given the magnitude in terms of size and sequential depth of many of these tasks, without the right set of algorithms, formal analysis would be infeasible. We in many cases needed to tune our algorithms to better scale to the necessary magnitudes for these applications. These formal solutions have made a substantial improvement to methodologies for verifying such pervasive logic for present and future designs.

While we have made significant strides in this direction, we note that pervasive logic verification is still far from a solved problem. Numerous pervasive verification tasks remain far outside the realm of proof capability, due to requiring the analysis of extremely large design components – possibly entire processor cores and even chips – for which sufficiently advanced algorithms are not available. Though traditionally addressed in distinct *test* conferences and conference tracks, we thus wish to introduce the challenges of pervasive logic verification to the formal verification community. In particular, we wish to encourage increased attention to applicable methodologies and continued research in the development of larger-capacity automated proof algorithms to formally address such pervasive-logic verification needs.

References

[1] D. Pham, "Key features of the design methodology enabling a multi-core SoC implementation of a first-generation Cell Processor," in *Asia and South Pacific Design Automation Conference*, Jan. 2006.

[2] S. Thakkar, "Second-generation Intel Centrino mobile technology," in *Intel Technology Journal*, vol. 9, Feb. 2005.

[3] J. Ludden et al., "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM Journal of Research and Development*, Jan. 2002.

[4] S. Chin, "IBM's eFuse technology portends adaptable chips," in *EE Times*, July 2004. http://www.eetimes.com/showArticle.jhtml?articleID=26100962.

[5] A. Orlowski, "The Cell chip - what it is, and why you should care," *The Register*, Feb. 2005. http://www.theregister.co.uk/2005/02/01/cell_analysis_part_one.

[6] IEEE Standards Board, *IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture*.

[7] M. Schwerdtfeger, "SPI - serial peripheral interface," June 2000. http://www.mct.net/faq/spi.html.

[8] Philips Semiconductors, *The I^2C-Bus Specification Version 2.1*. Jan. 2000. http://www.semiconductors.philips.com/acrobat_download/literature/9398/39340011.pdf.

[9] G. A. Van Huben, "The role of two-cycle simulation in the s/390 verification process," *IBM Journal of Research and Development*, vol. 41, no. 4/5, 1997.

[10] D. Das and N. A. Touba, "Reducing test data volume using external/LBIST hybrid test patterns," in *International Test Conference*, 2000.

[11] C. Stroud and G. Liang, "Design verification techniques for system level testing using ASIC level BIST implementations," in *IEEE International ASIC Conference and Exhibit*, 1993.

[12] D. Chang, M.-C. Lee, K.-T. Cheng, and M. Marek-Sadowska, "Functional scan chain testing," in *DATE*, Feb. 1998.

[13] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[14] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, Nov. 2000.

[15] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[16] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on CAD*, Dec. 2002.

[17] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, July 2006.

[18] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, March 1998.

[19] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *FMCAD*, November 2000.

[20] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *DAC*, June 2005.

[21] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.

[22] I.-H. Moon, H. H. Kwak, J. Kukula, T. Shiple, and C. Pixley, "Simplifying circuits for formal verification using parametric representation," in *FMCAD*, Nov. 2002.

[23] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *CHARME*, Oct. 2005.

[24] D. Wang, *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, May 2003.

[25] I.-H. Moon, G. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *FMCAD*, Nov. 2000.

[26] N. Een and N. Sörensson, "Temporal induction by incremental sat solving," in *BMC*, 2003.

[27] Accelera, *PSL LRM*. http://www.eda.org/vfv.

[28] C. T. Mo, C. L. Lee, and W. C. Wu, "A self-diagnostic BIST memory design scheme," in *Memory Technology, Design and Testing*, Aug. 1994.

[29] V. Schober, S. Paul, and O. Picot, "Memory built-in self-repair using redundant words," in *Int'l Test Conference*, Nov. 2001.

[30] E. Hallee, S.-L. Huang, K. Hwang, and B. Messina, "Fencing circuit and method for isolating spurious noise at system interface," in *U.S. Patent 5,386,393*, 1993.

[31] F. Injey, "External time reference (ETR) requirements on z990," in *IBM Redbooks Flash REDP-3753-01*, Feb. 2004.