Emulation in Post-Silicon Validation: It's Not Just for Functionality Anymore

(Invited Paper)

Kyle Balston*, Alan J. Hu†, Steven J. E. Wilton*, Amir Nahir‡

*Dept. of Electrical and Computer Engineering, University of British Columbia, Canada, {kyleb, stevew}@ece.ubc.ca
†Dept. of Computer Science, University of British Columbia, Canada, ajh@cs.ubc.ca
‡IBM Research, Israel, nahir@il.ibm.com

***Abstract*—FPGA-based emulation has emerged as an important tool in the overall validation process for an increasing number of large integrated circuits. Emulation gives the ability to validate a design using long-running, realistic tests, which are infeasible to perform using simulation. Traditionally, however, FPGA-based emulation has been used to validate only the *functional behavior* of an integrated circuit, since circuit-level properties (e.g., timing, noise margins, etc.) are obviously different between the FPGA emulation and the final integrated circuit. In this paper, we show that emulation can also be used as an important tool to assist validation of more than just functional behavior. In particular, we show how FPGA-based emulation can be used to evaluate the critical-path timing coverage of a validation plan, and show that the area and timing overheads are acceptable. We demonstrate this technique to measure the critical-path coverage of a complex SoC using common post-silicon validation tests — including booting Linux, and running targeted and random programs — giving valuable insight into the quality of such tests in covering the timing paths in a design post-silicon.**

I. Introduction

FPGA-based emulation has become a critical technology for the validation of large integrated circuits. Although designers make extensive use of software simulation when verifying their designs, simulation runs approximately eight orders of magnitude slower than actual silicon [14]. At this speed it would take more than three years to simulate just one second of silicon execution. FPGA-based emulation technology allows for much higher validation coverage by allowing designers to re-target their designs to one or more FPGAs and "execute" the design at orders of magnitude faster than simulation. For example, Intel's Atom processor was mapped to a single FPGA running at 50 Mhz [17], and their performance-oriented i7 Nehalem processor was mapped to five FPGAs running at 520 kHz [15]. While slower than the fixed-function chip, these speeds are fast enough to perform common validation tasks such as booting an operating system — tasks which are infeasible in simulation.

Because the underlying implementation is so different between the integrated circuit and the FPGA-based emulation, emulation typically captures only the functional behavior of the design. For example, when designers re-map their circuits to one or more FPGAs, they must compile their designs using the FPGA vendor tools, which may make different optimization decisions than the ASIC synthesis tools. In addition, the pre-fabricated nature of an FPGA means that certain structures (e.g., memories, multipliers) may be implemented very differently on an FPGA than on an ASIC. Furthermore, the physical implementation of a LUT is radically different from a standard cell or full custom gate meaning important properties like timing, critical paths, robustness to process and environmental variation, noise margins, etc. are all different in the emulated design. Hence, the traditional usage of emulation for validation is only for functional behavior; the conventional wisdom is that emulation cannot help with validating other properties.

In this paper, we show otherwise — emulation *can* be used as an important tool to assist validation of more than just functional behavior. In particular, we focus on timing validation and the task of quantifying the *critical-path coverage* of a validation plan. We define the critical-path coverage of a validation plan as the proportion of the critical and near-critical paths that are exercised during the execution of the plan. As will be discussed in Section II, measuring this quantity is essential, both to guide the creation of suitable post-silicon validation tests, and to determine how thoroughly a design has been validated before it is put into high-volume production. Our technique involves adding a small amount of instrumentation (*coverage monitors*) in strategic locations. Rather than add monitors to the actual silicon (which would have significant area overhead), we add them to an FPGA-based emulation of the design. By measuring coverage during emulation, we can obtain a metric for the timing coverage of our verification plan when applied to the actual chip. A key challenge is that the critical paths of the original ASIC may be different than the critical paths of the emulated version.

Note that our approach does not replace post-silicon timing validation, but instead measures how well it is being done. Nor are we performing timing analysis of FPGAs themselves [12], which would be different on the ASIC. Instead, we are measuring the quality of the post-silicon timing *tests*, by using FPGA emulation.

To summarize, this paper makes three contributions: (1) we propose a general methodology for using FPGA-based

This work was supported in part by Semiconductor Research Corporation contract 2010-TJ-2058.

978-1-4673-2899-9/12/$31.00 ©2012 IEEE

emulation to support post-silicon validation of more than just functional behavior, (2) we instantiate a specific instance of the methodology: using emulation to measure post-silicon critical-path coverage, and (3) we demonstrate the method on a complex SoC using common post-silicon validation tests, including booting Linux and running long random instruction streams, giving insight into how well such tests cover the critical timing paths in a design.

II. Critical-Path Coverage

Post-silicon validation encompasses everything that happens after the silicon of a new design is fabricated, but before the design is put into high-volume production, with the goal of catching critical design errors before producing large numbers of buggy chips. Post-silicon validation currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse [1], [7].

Many bugs caught in post-silicon validation are pure functional errors that ideally could have been caught pre-silicon. Other bugs, however, relate to electrical or physical properties of the chip and can only be truly validated in silicon. An obvious example is timing: how fast can the chip really run? What are the critical timing paths limiting speed improvements? Other examples include tolerance to voltage and temperature variations, electromagnetic interference, accelerated aging, etc. Many of these properties are also tested via timing tests, by varying some parameter and measuring at what frequencies the chip still behaves correctly, e.g., the classic "shmoo" plot obtained by running the chip at different voltages and measuring what frequencies give correct behavior.

For any of these tests to be meaningful, however, the tests must actually exercise the worst-case delays of the chip. If a test program fails to exercise the worst-case delays, a chip can erroneously pass the post-silicon timing tests, with the resulting design put into high-volume production only to fail in the field under real workloads. Accordingly, there needs to be a way to assure that a test actually exercises all the critical timing paths of a design. This is what critical-path coverage measures.

Designers regularly employ pre-fabrication timing verification techniques such as static timing analysis and statistical static timing analysis. These techniques provide valuable information regarding potential timing problems in a design, but are only approximate — importantly, the paths predicted to be critical do not correlate perfectly with the true critical paths [4], hence the need for expensive-but-more-accurate post-silicon timing validation [13].

The key challenges in post-silicon validation, however, relate to limited observability and controllability. Some commercial tools have emerged to aid in this task (e.g., [16]). However, regardless of the tools and techniques used to exercise the fabricated or emulated chip, it is necessary to know (a) how to create test cases that adequately exercise the chip, and (b) how to measure how complete these tests are. The latter question is known as coverage and is the focus in this paper. This is a critical question: without a method to quantify the quality of a given test set, it will be impossible to assert that a design has been adequately verified.

There has been some previous work proposing to add on-chip monitors to measure coverage information. In [6], the authors report post-silicon coverage monitoring on Intel's Core 2 Duo processor family, but due to the constraints of minimizing on-chip monitoring overhead, only very minimal coverage information was extracted. In [9], extensive on-chip monitoring for code coverage was added to an FPGA-based emulation of an SoC to measure the overhead imposed. In [2], the authors add coverage monitors for runs during pre-silicon emulation, giving a good estimate of the coverage achievable by the same tests post-silicon.

This previous work, however, did not explicitly quantify the coverage of specific *timing paths* in a design. In order to ensure correct timing behavior, it is necessary to ensure that the critical paths of a circuit are adequately exercised post-silicon. Our method quantifies and measures the effectiveness of a post-silicon verification plan in exercising long (therefore possibly critical, due to inaccuracies in pre-silicon timing analysis) paths in a circuit. The overall goal is: given a chip that has been fabricated, and a verification plan (software to run on the processor and other realistic input stimuli), determine to what extent the timing-critical paths of the circuit are exercised. More precisely, we seek to measure a coverage metric which we define to be the proportion of the n longest paths which have been exercised at least once during the execution of the verification plan. (In our experiments in Section VI, $n = 2048$, but n can be chosen arbitrarily.)

It is important to distinguish our approach from delay fault testing for manufacturing test. Many underlying concepts (e.g., path delay, sensitizability, critical paths, etc.) are the same, but the application is different. In high-volume manufacturing test, the design is assumed to be good, so the goal is only to catch manufacturing defects; however, test time per die must be kept extremely low, so tests must be short. In contrast, in post-silicon validation, the goal is to catch subtle design errors that escaped pre-silicon validation, so extremely long-running tests (e.g., billions of cycles of running OS and application software), with the die in an actual system, are used. Our goal is to be able to assess the quality of such long-running tests.[1]

III. Overall Flow

This section describes our methodology for obtaining delay coverage information for a circuit to be tested post-silicon, highlighting the role of FPGA-based emulation. Although our techniques could be applied to any large integrated circuit, we focus on large processor-based SoCs since the interaction

[1]Of course, in post-silicon validation, short targeted tests are also run on automated test equipment, but such tests are typically generated with coverage information known by construction or simulation, and therefore are not the focus of this paper. Conversely, Bernardi et al. [5] proposed an approach analogous to ours, but for delay-fault functional tests for manufacturing test. Because there is little benefit to emulating such short tests (which could easily be simulated), their approach essentially emulates an entire delay fault simulator, aggregating speed-up over many injected delay faults, at the cost of very large area overhead (roughly 3x slices in their experiments).

between hardware and software provide significant verification challenges.

One way to determine whether a path has been covered in hardware validation is to add small coverage monitors to each path of interest; during the run, these coverage monitors could trip if the corresponding path is exercised, and at the end of execution, the state of these monitors could be read, and the coverage metric calculated. The problem with this approach is the area overhead of the coverage monitors. Although these coverage monitors could be removed in the release version of the chip, doing so would change the timing and electrical behavior of the chip, likely leading to different paths becoming critical, nullifying the value of the results.

Our approach takes advantage of the prevalence of FPGA-based emulation. In the following discussion, we use the term *ASIC version* of the design to refer to the version of the design before emulation (the version that will eventually be implemented as a chip and shipped) and the term *emulation version* of the design to refer to the version of the design that has been re-mapped to FPGAs for validation and prototyping purposes.

Our flow uses the *emulation version* to estimate the timing coverage metric of the *ASIC version*. More precisely, our approach is as follows. Using static timing analysis, we identify the n longest paths in the ASIC version. If information regarding false paths is available, those paths can be discarded at this stage. Similarly, if there are other paths of particular interest to the design and validation engineers, they can be added at this stage. Once we have identified the paths of interest in the original design, we create the emulation version by mapping the circuit to FPGAs (e.g., as described in [17], [15]). We then instrument the emulated design, as will be described in Section IV. This instrumentation monitors the targeted paths, and determines when each has been exercised. At the end of the run, the coverage information can be read from the instrumentation, and an overall timing coverage metric calculated. This is the coverage that is expected when the same verification plan is run on the ASIC version of the design. The proposed flow is shown in Figure 1.

In the above flow, it is important that the paths be determined using the original design, since once the design has been re-targeted to an FPGA for emulation, the paths that are critical may change. It is very possible that the paths we target are *not* the critical paths during emulation, but they are the ones we care about, since they are the critical paths of the chip that we intend to ship.

IV. Instrumentation

Key to the flow described in Section III is the instrumentation that is added to the emulation version of the circuit. Because we are adding our instrumentation to the emulation version of the circuit only, the area and delay overheads do not translate to the shipped chip. However, minimizing overhead is still important, since the overhead directly relates to the size of the FPGA emulation system and the speed at which emulation tests can be run.

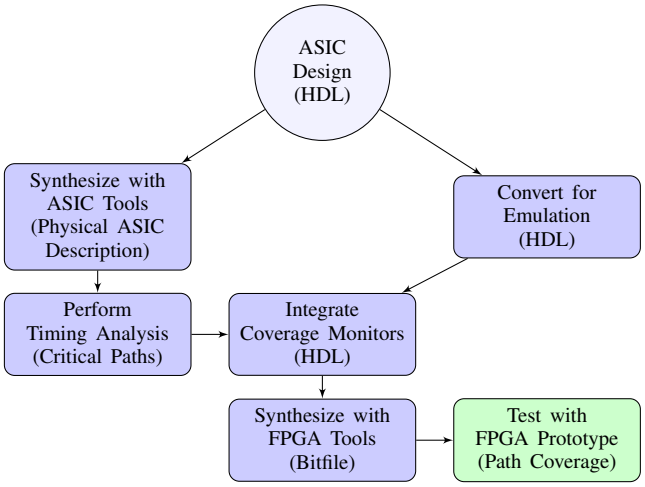

Fig. 1. Toolflow for adding path coverage monitors to the FPGA prototype of an ASIC design. Each stage's output is shown in parentheses.

As described in Section III, we obtain a set of timing-critical paths from the static analysis of the ASIC version of the circuit, and then instrument these paths in the FPGA version of the circuit. The instrumentation is designed to record whether, for each path, the path has been "exercised". A monitored path is considered exercised, and therefore covered, when a transition on the input of the path causes there to be a transition along every element of the path. To instrument a path, we add (a) a transition detector at the start of the path, (b) logic to calculate (by Boolean difference [8]) the sensitization condition [3] for each logic element along the path, and (c) a bit to record whether the path has been entirely sensitized. An example is shown in Figure 2, where the logic function, f, is $(\overline{A+B}+C)\cdot(\overline{A}+\overline{B})$. From this logic, its sensitization function can be calculated to be $B \cdot C$. This makes intuitive sense as when B = '1' the upper OR gate's $\overline{B}$ input will be '0', allowing signal A to control that gate's output. This is similarly true for the final AND gate; when C = '1' its lower input will be '1', allowing the targeted path to control the value of f. Figure 2(a) shows the original path, and Figure 2(b) shows the instrumented circuitry, including the logic to compute the sensitization condition.

It is important that we add instrumentation circuitry to the RTL representation of the circuit, before it is optimized and mapped by the FPGA tools. It is very possible that after mapping to the FPGA, the original paths either do not exist (because of the optimizations performed by the FPGA tools) or because parts of the original paths have been encapsulated into hard blocks (such as DSP blocks). Note, however, that the inclusion of this instrumentation does not prevent the FPGA tool from optimizing the original circuit as it sees fit. In the example of Figure 2, once the instrumentation has been added to the RTL version of the circuit, the tool is free to optimize the original path in any way (including removing it from the circuit entirely). The semantics of these monitors will be

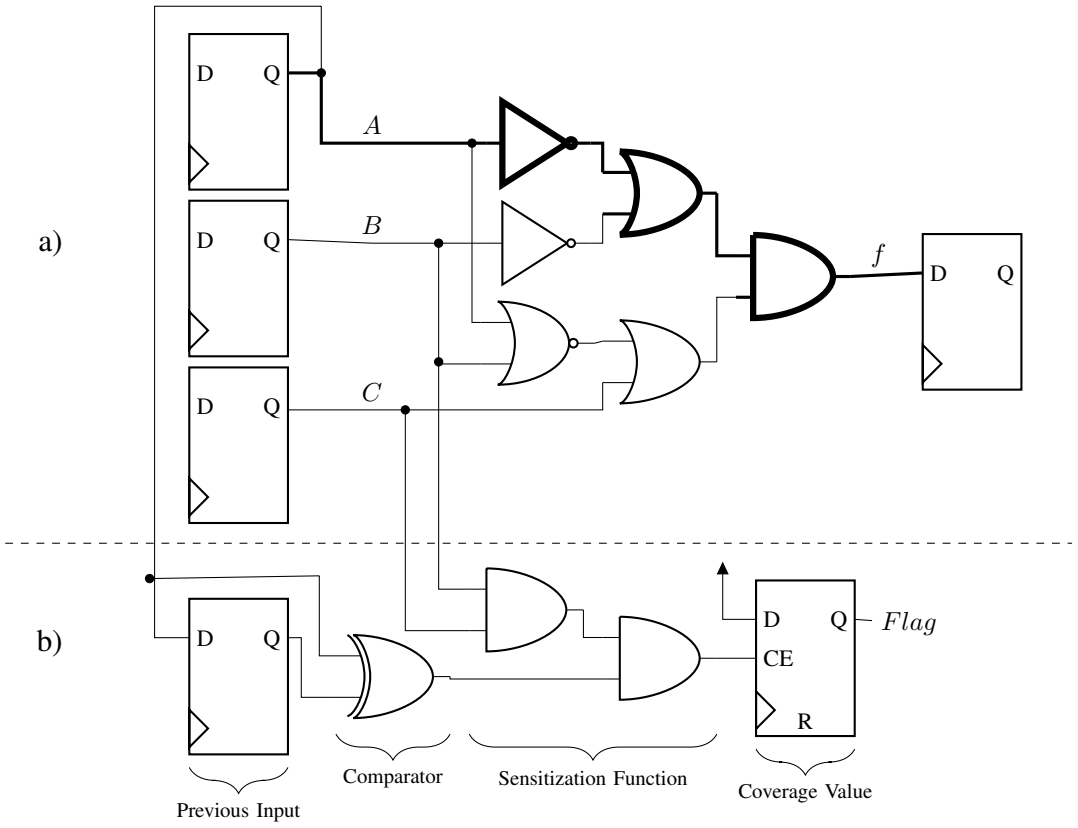

Fig. 2. Example of a critical path, a), instrumented to monitor coverage by b) for bolded path f through input A.

preserved by the tools even if that original path does not exist in the emulation version.

The area overhead of this instrumentation is proportional to the number of paths instrumented and the length of each path. For each segment along the path, the sensitization function is guaranteed to require no more extra lookup tables than a gate-level implementation of the original path, since the sensitization function at each stage requires fewer inputs than the original function at each stage. However, in some cases, it may be that the original function can be optimized effectively by the FPGA CAD tools (for example, a multiply function may be efficiently mapped to an embedded DSP block), while the instrumentation logic can not be optimized as efficiently. In the worst case, if the instrumentation circuitry requires signals that would otherwise have been encapsulated inside a DSP block, then the circuitry generating these signals will have to be replicated in the FPGA soft logic. If this occurs, it may increase the overhead of our instrumented circuit, compared to the same circuit without instrumentation, beyond that which would be expected due to the extra logic. However, as we will show in Section VI, experimentally, this rarely occurs.

The delay overhead in the emulation version is very small. Although some signals will have increased fanout, because FPGA interconnect is typically fully buffered [11], the overall timing impact will be small.

V. Methodology

A. Approximated Flow

Because the focus of this work is the design and implementation of the coverage monitors introduced in Section IV, as well as the investigation of the coverage achieved for some common post-silicon tests, and not the verification of a processor-centric SoC *per se*, we have shortened the proposed verification flow in two ways: 1) by starting with a design that is already known to be synthesizable to FPGA, and 2) by integrating coverage monitors into the design at the gate-level, not the RTL level.

Starting with a design that is already synthesizable to an FPGA, particularly one that can be fabricated to ASIC technology, is a very good approximation to using an FPGA-emulated ASIC design. However, it causes our flow to differ in one significant way: there is no ASIC design from which to obtain timing paths. Instead, we use timing results from the design natively synthesized for the FPGA. This modified flow is shown in Figure 3. Obviously, the specific timing paths from the FPGA timing analysis will likely differ from what would have been obtained for an ASIC version, but they will not be qualitatively different: the FPGA timing paths used in our approximated flow could be easily exchanged for ASIC timing paths. It is important to highlight that this simplification was made to facilitate our experimentation; in a "real" application

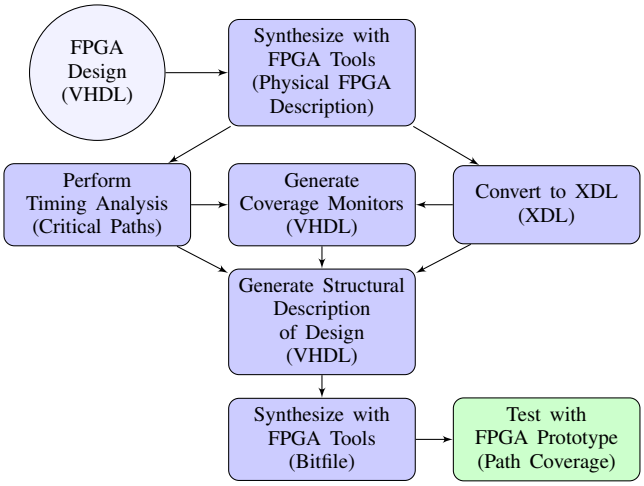

Fig. 3. Toolflow used in this paper to automatically add path coverage monitors to an FPGA design. Each stage's output is shown in parentheses.

of the flow, ASIC timing paths would be used.

We integrated our coverage monitors at the gate-level because, to the best of our knowledge, there are no full-featured, open-source VHDL parsers available for adaptation to our uses. In contrast, the two biggest FPGA vendors, Altera and Xilinx, both provide toolkits which allow experimental integration with their tools. For this paper, we used the RapidSmith API from BYU [10] for parsing and iterating over the Xilinx Design Language (XDL) format, Xilinx's human readable gate-level circuit description. Parsing XDL is much easier than parsing VHDL because it is a simple, structural description of a circuit written by an automated tool and not a language created for circuit designers to write directly.

B. Automated Integration of Coverage Monitors

A key aspect of our ideal and approximated flows is the two CAD tool passes: the design first must be synthesized, placed, and routed (ideally, with the ASIC tools) in order for static timing analysis to determine likely critical paths of the ASIC version; then this design is instrumented for coverage and processed a second time by the FPGA tools to create the emulation version. Accordingly, to automate the flow, we need a mechanism to extract the output from the first step, add instrumentation, and then input this into the FPGA tools for the second step.

Xilinx provides unofficial support for an interface to export their proprietary Netlist Circuit Description (NCD) files to a human-readable XDL file in terms of their fundamental FPGA primitives and the nets that connect them. This allows designs to be imported and exported freely between the major FPGA synthesis stages: *mapping*, *placement* and *routing*. However, in recent versions of the Xilinx tools, the *mapping* and *placement* stages have been integrated, making import/export after mapping but before placement impossible. In these versions, it is possible to export a placed description and discard the placement information, but there is no way to pass a mapped-but-unplaced description to the Xilinx placer.

XDL can describe an unplaced, a placed-but-not-routed, or a placed-and-routed physical description of a circuit on a Xilinx FPGA. These XDL files are composed of *instances* which represent the Xilinx building blocks such as slices, input/output buffers, DSPs and BRAMs, and *nets* which describe how they are connected together. Listing 1 shows a sample instance with net connections for a combinational slice which implements the $\overline{A_2} \cdot A_6$ function in the A lookup table (described by the slice's A6LUT attribute).

We use this netlist information to generate an equivalent VHDL description of the XDL file alongside additional monitoring logic. This new VHDL description will be be functionally equivalent to the original design but will include our coverage monitors. This task is made possible by instantiating Xilinx FPGA primitives through the VHDL libraries provided by Xilinx. However, in most cases the XDL describes the FPGA primitives differently than can be directly instantiated in VHDL. For example, XDL describes combinational logic in terms of lookup table attributes within a slice instance, but the Xilinx libraries must instantiate slices in terms of their components (such as lookup tables, muxes, carry chains, etc). Similarly, the BRAM primitives that XDL describes include pins which are held at GND or VCC but the BRAM instances that can be instantiated in VHDL only include ports which are used. For example, a RAMB18X2SDP instance's RDADDR and WRADDR vectors are 6 signals wider as described in XDL than can be instantiated in VHDL.

In general, this process is fairly mechanical. However, it requires careful analysis to ensure that each instance is converted correctly. We used an industrial formal equivalence tool to ensure that the VHDL description generated from the XDL was equivalent to the original circuit.

C. Implementation

To demonstrate that our overall flow works, we applied it to a substantial system-on-chip design. The system-on-chip that we have used is composed of the open-source LEON3 SPARCv8 processor, provided by Aeroflex Gaisler, and various peripherals supplied by the open-source community. It is roughly comparable to a netbook or tablet device and can be fabricated to 0.13 µm ASIC technology at a speed of 400 MHz. Its maximum operating frequency on our Virtex-5 XC5VLX110T FPGA device is 80 MHz, a typical frequency for FPGA-based designs. The processor is fast enough to boot Linux and run the X11 windowing system.

We selected the four IP blocks shown in Table I to instrument for timing-path coverage. All IP blocks were available in VHDL.

VI. Results

Path coverage results for the four IP blocks shown in Table I were obtained for the top 2048 most critical paths, as determined by the Xilinx Timing Analyzer, for the tests shown in Table II. These tests were chosen to be representative of

```
inst 'eth1.e1/m100.u0/ethc0/r.rxwriteack' 'SLICEL',
placed CLBLM_X33Y57 SLICE_X57Y57,
cfg '
// Shared slice attributes
CEUSED::0 // no chip enable
CLKINV::CLK // clock is not inverted
PRECYINIT::#OFF // no Cin for carry chain
REVUSED::#OFF // no reverse set/reset signal
SRUSED::0 // no set/reset signal

// Attributes for lookup table (LUT) A
A5LUT::#OFF // slice is not in dual 5LUT mode
A6LUT:eth1.e1/m100.u0/ethc0/r_rxwriteack_cmp_eq0000
     :#LUT:O6=(~A2*A6) // not(A2) and A6
ACY0::#OFF // no AX passthrough or 5LUT needed
AFF:eth1.e1/m100.u0/ethc0/r.rxwriteack:#FF // output
     of 6-input LUT goes through flip-flop
AFFINIT::INIT0 // initial value of flip-flop is '0'
AFFMUX::O6 // Input to FF is output of  6LUT
AFFSR::SRLOW // set/reset signal in normal mode
AOUTMUX::#OFF // no additional AMUX signal needed
AUSED::#OFF'; // Non-registered output not used
// Lookup tables B, C and D are omitted for brevity.

// Nets: Non-relevant inpins are omitted
net 'eth1.e1/m100.u0/ethc0/r.rxwriteack',
  outpin 'eth1.e1/m100.u0/ethc0/r.rxwriteack' AQ,
  inpin 'eth1.e1/m100.u0/ethc0/r.rxwriteack' A6;

net 'eth1.e1/m100.u0/N300',
  outpin 'eth1.e1/m100.u0/N300' B,
  inpin 'eth1.e1/m100.u0/ethc0/r.rxwriteack' A2;
```

Listing 1. Representation of a SLICEL instance in XDL along with its nets.

TABLE I
IP BLOCKS UNDER TEST

IP Block	Lines of RTL	Description
ddr2spa	1924	DDR2 Controller
iu3	3131	7-stage Pipelined Integer Unit
mmu	1929	Memory Management Unit
mul32	24477	(Un)signed 32-bit Multiplier

TABLE II
POST-SILICON VALIDATION TESTS USED IN EXPERIMENTS

Name	Description
Dhry	Dhrystone is a synthetic integer benchmark used to calculate general processor performance
Hello	The classic "Hello, World" program consisting of a single printf
Linux	Booting an embedded operating system; Operating System built using BuildRoot 2012.02 with Linux 3.0.4
Random	A series of 10,000 random programs generated by Csmith [18]
Stanford	Eight small benchmarks including integer matrix multiply, sorting and permutation algorithms
Systest	Pre-silicon system-level tests for the LEON3 processor, supplied by Aeroflex Gaisler

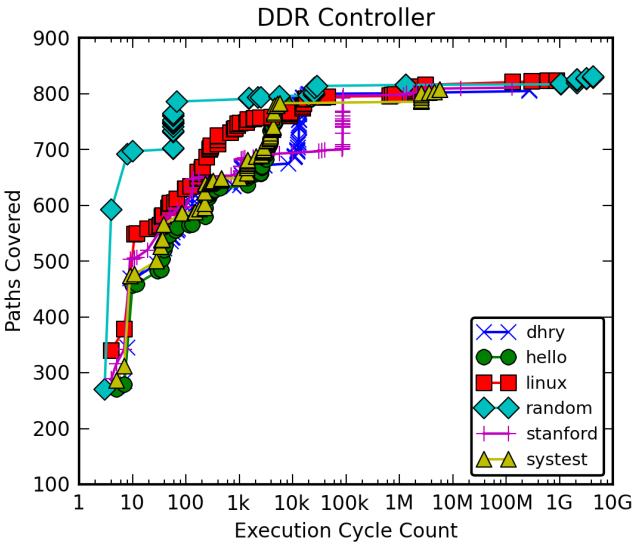

Fig. 4. Coverage achieved over time for the DDR controller (ddr2spa).

what is generally run during the post-silicon stage of testing: various long-running system-level tests, random tests, and application software. Included for comparison is the set of pre-silicon system-level tests supplied by Gaisler for the LEON3 processor.

Table III shows the coverage achieved for each test on each block, and Figures 4–7 plot the coverage obtained over time on these blocks. The dominant overall trend is that more cycles translates into better coverage, hence emphasizing the value of long-running post-silicon tests. For example, the Linux boot tends to achieve the highest coverage, but the long-running `random` benchmark gradually catches up. We also see that coverage often comes in bursts, when an application starts a new phase of computation that performs different operations. There are also some peculiarities, e.g., the `random` benchmark does particularly well on the DDR controller. We believe this is because the benchmark is loading in a series of programs one-after-another from memory, so it exercises the DRAM extensively very early on. Similarly, the Stanford benchmark does particularly well on the multiplier, presumably due to the matrix multiplication program in the benchmark suite.

TABLE III
PATH COVERAGE RESULTS (2048 MOST CRITICAL PATHS)

IP Core	Dhry	Hello	Linux	Random	Stanford	Systest
ddr2spa	807	785	811	807	831	824
iu3	225	135	330	257	231	201
mmu	1210	508	1431	1351	1288	1063
mul32	175	71	271	325	334	247

Beyond overall coverage, the monitors give detailed path-level coverage information — we know exactly which paths were covered, and when. For example, an obvious question a validation engineer would ask is whether any test is dominated by the others. The coverage monitors allow us to answer this question. For example, Table V shows, for each test, how many paths were covered by that test, but not by some other test. For example, looking at the first row, second column, we see that

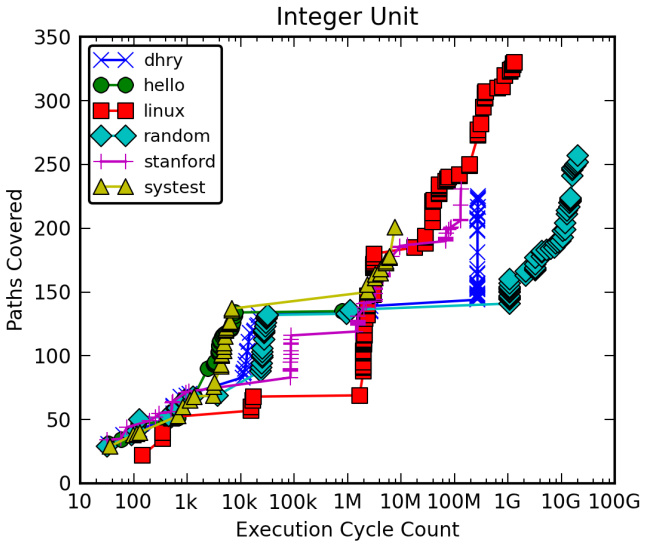

Fig. 5. Coverage achieved over time for the integer unit (iu3).

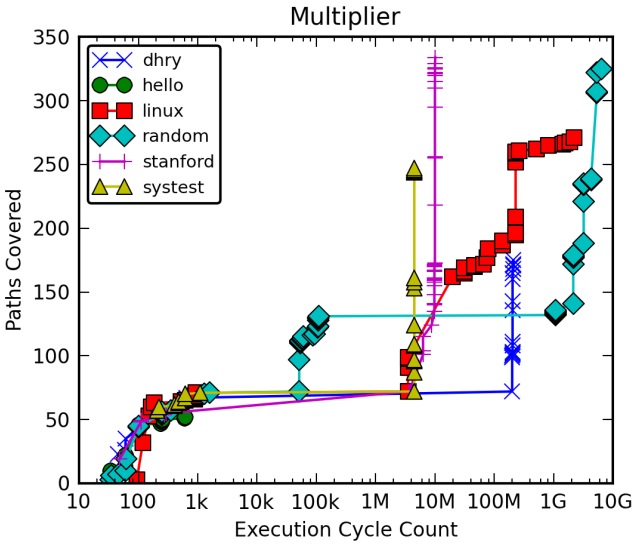

Fig. 7. Coverage achieved over time for the multiplier (mul32).

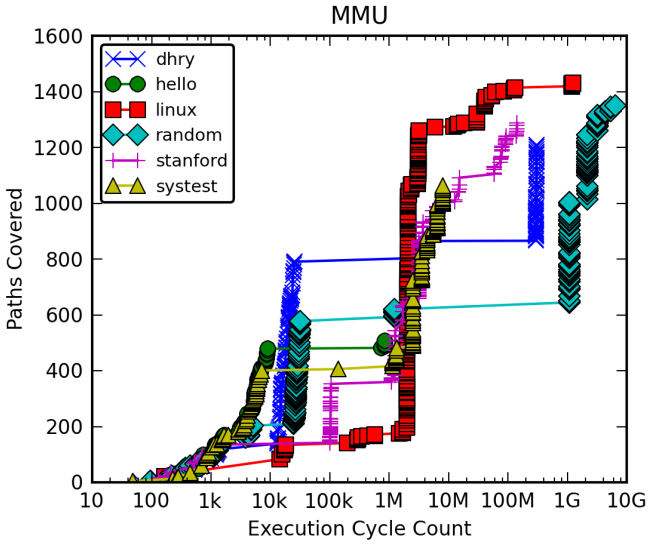

Fig. 6. Coverage achieved over time for the MMU (mmu).

`dhry` covered 112 paths that `hello` did not. The last row and column compare each test to all the others. For example, the last column shows that `linux`, `random` and `systest` all covered paths which all other tests missed. In practice, this sort of information is invaluable: it identifies coverage holes and redundancies in the test plan, and can also help in *diagnosing* problems, e.g., if tests that cover a certain path fail, but tests that don't cover that path pass.

Table IV shows the area overhead of our instrumentation, in FPGA LUTs. More informative is to plot the area overhead per monitored path. We generated monitors for 512, 1024, 1536, as well as 2048 paths, with the results shown shown in Figure 8. From the figure, we see that the area per monitored path is roughly constant, just as predicted. Indeed, it tends to drop off slightly, due to a combination of shorter paths as we monitor less critical ones, and greater sharing and optimization between paths being monitored. This shows that we can always scale back the number of paths monitored if emulation area overhead becomes problematic.

TABLE IV
AREA OVERHEAD TO MONITOR 2048 CRITICAL PATHS

IP Block	Average Path Length (Xilinx primitives)	Area Overhead (Slices)
ddr2spa	2.5	2654
iu3	7.4	1430
mmu	3.2	1946
mul32	14.6	2918

The monitoring circuitry did not impact the specified SoC timing — the FPGA attained the same frequency.

VII. CONCLUSION

We have presented a novel emulation-based approach for assessing the quality of post-silicon timing validation tests. In our experiments, we have demonstrated the ability of our approach to measure the coverage achieved by a variety of typical post-silicon validation tests on a non-trivial SoC. The timing overhead (impacting the speed of emulation) of the approach is negligible, and the area overhead is controllable, being proportional to the number of paths monitored.

Specifically for timing validation, there are several promising avenues for further research. First, it would be useful to prune the paths reported by the static timing analyzer for functional reachability. This would make the coverage metric more accurate. Secondly, although area overhead is not a principle concern for emulation, reducing the area overhead further

TABLE V
COVERAGE ACHIEVED BY ONE TEST BUT NOT BY ANOTHER

		...which was not achieved by this test						
		Dhry	Hello	Linux	Random	Stanford	Systest	All Others
Coverage achieved by this test...	Dhry	0	112	2	21	8	60	0
	Hello	0	0	2	0	3	8	0
	Linux	124	236	0	96	116	152	52
	Random	77	168	30	0	65	105	25
	Stanford	18	125	4	19	0	67	0
	Systest	36	96	6	25	33	0	3
	All	155	267	33	99	145	179	

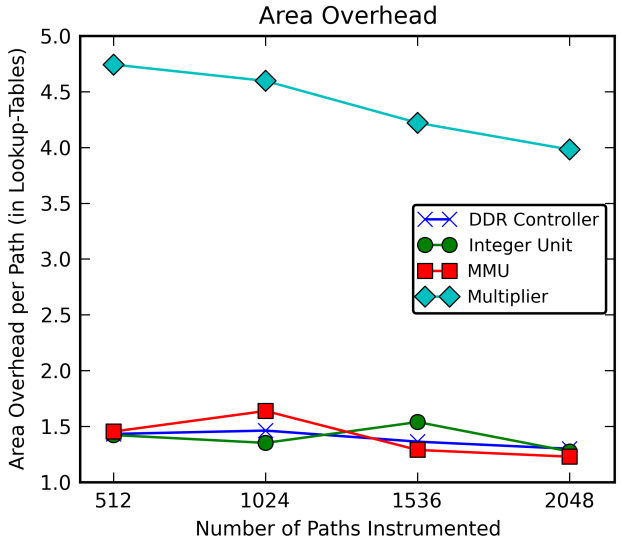

Fig. 8. Area overhead per path as the number of targeted paths increases.

would allow more paths to be monitored. Although there are certain straightforward but time-consuming optimizations that could be made to reduce area overhead, it would be interesting to investigate whether there are area reductions specific to coverage monitors. Thirdly, our current monitors record only whether a path has been sensitized; additional monitors could be added to record whether the result is observable. Finally, coverage information could be used as a feedback mechanism to help develop better tests, e.g., via genetic algorithms.

More generally, our work shows that FPGA-based emulation can aid more than just functional validation. There is considerable potential to use our approach to extend emulation to other validation tasks.

VIII. PUBLIC DISTRIBUTION

The XDL translation scripts are publicly available at http://ece.ubc.ca/~kyleb/files/xdl_translation_scripts.zip

REFERENCES

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Design Automation Conference*, pages 7–12. ACM/IEEE, 2006.

[2] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann. Reaching coverage closure in post-silicon validation. In *Haifa Verification Conference*, pages 60–75. Springer, 2011.

[3] D. B. Armstrong. On finding a nearly minimal set of fault detection tests for combinational logic nets. *IEEE Trans. on Elec. Comp.*, EC-15(1):66–73, Feb. 1966.

[4] P. Bastani, B. N. Lee, L.-C. Wang, S. Sundareswaran, and M. S. Abadir. Analyzing the risk of timing modeling based on path delay tests. In *Intl. Test Conference*. IEEE, 2007.

[5] P. Bernardi, M. Grosso, and M. Sonza Reorda. Hardware-Acclerated Path-Delay Fault Grading of Functional Test Programs for Processor-Based Systems. In *GLSVLSI2007: 17th ACM Great Lakes Symposium on VLSI*, pages 411–416, 2007.

[6] T. Bojan, M. Arreola, E. Shlomo, and T. Shachar. Functional Coverage Measurements and Results in post-Silicon validation of Core 2 Duo family. In *High Level Design Validation and Test Workshop*, pages 145–150. IEEE, 2007.

[7] K.-H. Chang, I. L. Markov, and V. Bertacco. Automating Post-Silicon Debugging and Repair. In *Computer-Aided Design*, pages 91–98. IEEE/ACM, 2007.

[8] A. C. L. Chiang, I. S. Reed, and A. V. Banes. Path sensitization, partial boolean difference, and automated fault diagnosis. *IEEE Trans. on Comp.*, 21(2):189–195, Feb. 1972.

[9] M. Karimibiuki, K. Balston, A. Hu, and A. Ivanov. Post-silicon Code Coverage Evaluation with Reduced Area Overhead for Functional Verification of SoC. In *High Level Design Validation and Test Workshop*, pages 92 –97. IEEE, Nov. 2011.

[10] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings. Rapid Prototyping Tools for FPGA Designs: RapidSmith. In *Field-Programmable Technology*. IEEE, Dec. 2010.

[11] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and Single-Driver Wires in FPGA Interconnect. In *Field-Programmable Technology*, pages 41–48. IEEE, 2004.

[12] J. Levine, E. Stott, G. Constantinides, and P. Cheung. Online Measurement of Timing in Circuits: For Health Monitoring and Dynamic Voltage & Frequency Scaling. In *Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2012.

[13] S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Design Automation Conference*, pages 12–17. ACM, 2010.

[14] A. Nahir, A. Ziv, and S. Panda. Optimizing Test-Generation to the Execution Platform. In *Asia and South Pacific Design Automation Conference*, pages 304 –309. IEEE, Feb 2012.

[15] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang. Intel Nehalem Processor Core Made FPGA Synthesizable. In *FPGA*, pages 3–12. ACM/SIGDA, 2010.

[16] Tektronix. Clarus SoC Post-Silicon Validation Solution. http://www.tek.com/embedded-instrumentation/clarus-soc-post-silicon-validation-solution, 2012.

[17] P. Wang, J. Collins, C. Weaver, B. Kuttanna, S. Salamian, G. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, et al. Intel Atom Processor Core Made FPGA-Synthesizable. In *FPGA*, pages 209–218. ACM/SIGDA, 2009.

[18] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294. ACM/SIGPLAN, 2011.