Leveraging Reconfigurability to Raise Productivity in FPGA Functional Debug

Zissis Poulos[1], Yu-Shen Yang[2], Jason Anderson[1], Andreas Veneris[1], Bao Le[1]

[1]Dept. of ECE, University of Toronto, Toronto, Canada. {zpoulos, janders, veneris, lebao}@eecg.utoronto.ca
[2]Vennsa Inc., Toronto, Canada, terry@vennsa.com

***Abstract*—We propose new hardware and software techniques for FPGA functional debug that leverage the inherent reconfigurability of the FPGA fabric to reduce functional debugging time. The functionality of an FPGA circuit is represented by a programming bitstream that specifies the configuration of the FPGA's internal logic and routing. The proposed methodology allows different sets of design internal signals to be traced *solely* by changes to the programming bitstream followed by device reconfiguration and hardware execution. Evidently, the advantage of this new methodology vs. existing debug techniques is that it operates without the need of iterative executions of the computationally-intensive design re-synthesis, placement and routing tools. In essence, with a single execution of the synthesis flow, the new approach permits a large number of internal signals to be traced for an arbitrary number of clock cycles using a limited number of external pins. Experimental results using commercial FPGA vendor tools demonstrate productivity (i.e. run-time) improvements of up to 30× vs. a conventional approach to FPGA functional debugging. These results demonstrate the practicality and effectiveness of the proposed approach.**

I. Introduction

As the cost of state-of-the-art ASIC design continues to escalate, field-programmable gate arrays (FPGAs) have become widely used platforms for digital circuit implementation. FPGAs carry several advantages over ASICs, including reconfigurability and lower NRE costs for mid-to-high volume applications. While there remains a gap between FPGAs and ASICs in terms of circuit speed, power and logic density [1], innovations in FPGA architecture, circuits and CAD tools have produced steady improvements on all of these fronts. Today, FPGAs are a viable target technology for all but the highest volume or low-power applications.

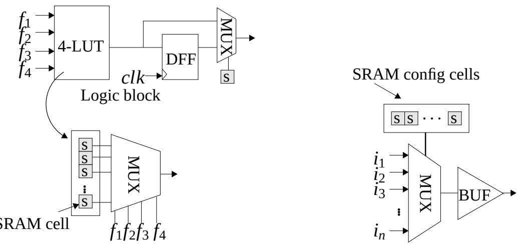
The reconfigurability property of FPGAs reduces the cost associated with fixing the various functional errors that can occur during the design cycle. With ASICs, designers spend considerable time in simulation/verification before tape out, including, for example, simulation with post-layout extracted capacitances and cross-talk noise analysis. Conversely with FPGAs, designers rarely do post-routing full delay simulations. Instead, reconfigurability allows design iterations to include actual silicon execution. Designers verify their design in hardware using the same (or a similar) FPGA they intend to deploy in the field. When design errors are discovered, the design's RTL is altered, re-synthesized and executed in hardware.

The time needed for design cycles in FPGAs is dominated by re-synthesis (logic synthesis, technology mapping, placement and routing) tool run-times. FPGA placement and routing can take hours or days for the largest designs [2], and such run-times are an impediment to designer productivity. With this observation in mind, in this paper, we present new techniques for FPGA functional debug that exploit the reconfigurability concept to raise productivity by reducing the number of compute-intensive design re-synthesis runs that are needed.

At a high-level, our approaches work as follows: Say, for example, an engineer wishes to trace a large number, N, of a design's internal signals during functional debug, using a small number of available external pins, m ($N >> m$). We augment the design with additional circuitry that allow the N signals to be traced with $\lceil N/m \rceil$ FPGA device re-configurations and hardware executions. The key value of our approach is that the design is only synthesized, placed and routed *once*, rather than $\lceil N/m \rceil$ times. This is achieved by selecting the different sets of m trace signals through modifications to the FPGA's configuration bitstream (i.e. the post-routed design).

While the proposed approach leverages reconfigurability to reduce loops through the design process, a further contribution of this work is a new multiplexer (MUX) design scheme for FPGAs that uses significantly less area than a traditional MUX design. The new MUX is suitable for use in cases wherein the MUX select inputs are changed using the FPGA bitstream, instead of using normally routed logic signals. We also present a design variant to handle the scenario where limited external pins are available for debugging.

As compared with design re-synthesis for each group of m signals, experimental results demonstrate that our approach improves run-time by up to 30×. Our approach also offers stability in the timing characteristics of the circuit being debugged.



(a) FPGA logic structures (b) Routing structures

Fig. 1. FPGA hardware structures.

The remainder of this paper is organized as follows. Section II reviews background on FPGA architecture and related work on FPGA functional debug. The proposed approach to debugging is described in Section III, as well as various architectures to meet different resource constraints. Section IV provides experimental results. Conclusions and suggestions for future work are offered in Section V.

II. Background

A. FPGA Architecture

An FPGA is a two-dimensional array of programmable logic blocks and a configurable routing network. Combinational logic functions in FPGAs are implemented using K-input look-up-tables (LUTs), which are small memories capable of implementing *any* logic function of up to K variables. As shown in Fig. 1(a), each LUT in an FPGA logic block is normally coupled with a flip-flop, which can optionally be bypassed. SRAM configuration cells are programmed to specify the truth table of the logic function implemented by the LUT, as well as control the flip-flop bypass MUX.

Fig. 1(b) shows a simplified view of a programmable routing structure. The inputs to the MUX attach to logic block output pins or routing conductors in the FPGA device (metal wire segments). The output of the buffer can drive a routing conductor or a logic block input. Again, SRAM configuration cells drive the select inputs on the MUX, and the SRAM values specify a particular input whose signal is driven through the buffer.

Fig. 1 is intended to illustrate that the logic functionality and routing connectivity of an FPGA depends entirely on values in the programming bitstream that is shifted into the FPGA's SRAM configuration cells (which are connected in a scan chain). The programming bitstream also specifies the initial value (logic-`0` or logic-`1`) for each flip-flop in the device. Our approaches to FPGA functional debug rely on making changes to the programming bitstream, without having to re-run time-consuming FPGA synthesis, place and route tools.

B. FPGA Functional Debug

There are two major approaches to perform functional debug with an FPGA. The first approach is to implement the complete design in an FPGA device. This is suitable for small designs that do not need to be executed at a high frequency. Because of the reconfigurability, debugging modules can be easily added or modified with no cost. A set of circuit modifications that enhance debug capability is presented in [3]. It provides software-like debug features, such as watchpoints and breakpoints. However, any modification to watchpoints or breakpoints requires recompilation of designs – a run-time intensive task. In a somewhat similar manner to what is proposed in this work, Graham et al. improve debugging productivity by instrumenting FPGA bitstreams [4]. An embedded logic analyzer is inserted into the design without connecting to any signals. After place-and-route, the signals targeted for tracing are routed to the logic analyzer by modifying

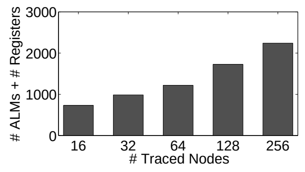


Fig. 2. Area overhead of SignalTap.

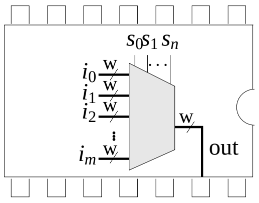


Fig. 3. Multiplexer for signal selection.

bitstreams using vendor tools. Although the approach provides flexibility in choosing the desired internal signals for tracing, it remains a very complicated procedure. Furthermore, when different sets of signals are selected for tracing, re-routing needs to be performed, which can significantly affect the timing closure of the design.

Xilinx's *ChipScope* tool provides features to trace different signals without re-executing place and route [5]. Special logic analyzer hardware is inserted into a design to trace internal signals during design execution. The captured signals can then displayed using Xilinx's ChipScore Pro Analyzer Tool, running on a connected host computer. While the approach bears similarity to our own, the techniques and tools associated with ChipScope are proprietary and not disclosed publicly.

The second approach to using FPGAs for functional debug is that of embedding reconfigurable logic into SoCs to enhance debug capability [6], [7]. The programmability of reconfigurable logic can be applied to implement various debug paradigms, such as assertions, signal capture and what-if analysis. Those paradigms help engineers to understand the internal behavior of the chip and provide at-speed in-system debug. Engineers can instrument the reconfigurable logic on-the-fly, as needed. However, each change to the debug circuitry incurs significant cost and overhead.

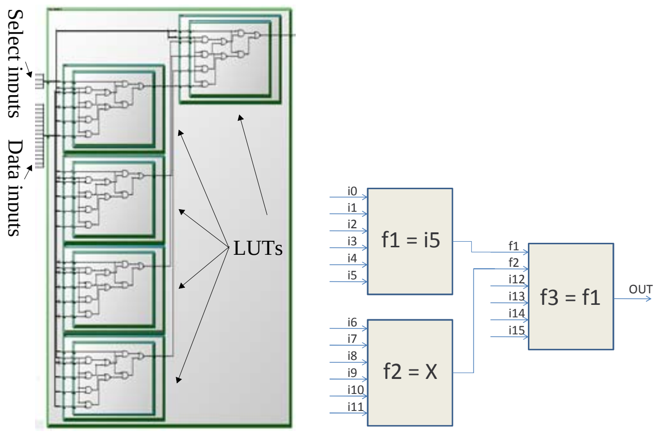
Finally, several works on selecting the signals that one may wish to trace for debugging have been proposed [8], [9], [10]. While most works target ASIC designs, the work in [10] is designed specifically for FPGAs. It predicts which signals may be useful for debugging and automatically instruments the design. Any prior work on signal selection could be used in conjunction with our approach.

III. A Reconfigurability-Driven Approach to FPGA Functional Debug

This section presents a new approach to enhance the observability of FPGA designs for functional debug. To debug functional errors in an FPGA design, the design is first synthesized, placed and routed on the target FPGA device. The programming bitstream is generated, programmed into the FPGA, and execution commences. If unexpected behavior is observed, a set of internal signals is selected to be traced by a logic analyzer to provide more information. In the conventional debug process, the design needs to be recompiled and the FPGA needs to be reprogrammed. Fig. 2 shows the area overhead of Altera's SignalTap [11] logic analyzer vs. the number of signals being tapped. One can see that the overhead grows significantly as the number of monitored signals increases. Due to the area overhead of the logic analyzer, usually only a small set of signals are traced at any one time. The process is repeated until the values for all signals of interest are acquired. The main issue with this process is that it can take hours to compile large designs [12]. As such, repeated compilation can introduce significant time overhead and prolong the overall debug process.

To alleviate the issue, a new design process that avoids recompilation is presented in this work. The idea is to modify the bitstream directly when different signals need to be traced. This is achieved by inserting a multiplexer into the design implemented on the FPGA, with the MUX inputs being all signals that one potentially wants to trace. Fig. 3 depicts a multiplexer that can select one of `m` groups of `w` signals. The select signals of the multiplexer are preset to logic-`0` or to logic-`1`. Then, one can trace different signals by manipulating the bitstream to set the select signals to different constants. Since there is no re-routing required, the bitstream modifications can be done easily. As a result, the time overhead of this process is reduced to a bitstream modification followed by a bitstream downloading. Bitstream downloading normally requires only seconds – significantly less overhead than the re-compilation approach.

Another advantage of the proposed process is its negligible effect on the stability of the design. In the conventional debug process, the design is re-routed each time when different signals are selected, As a result, designers often need to readjust the design to meet the various timing constraints. Even though recent FPGA tools provide incremental compilation to preserve the engineering efforts from previous place/route steps, experiments show that the speed performance of designs after incremental compilation can vary. In the proposed process, because all signals one potentially wants to trace are connected to the selection module at the beginning, only the original one compilation is necessary. As a result, selecting different signals through bitstream modifications minimizes the overall impact on the performance of the design. Note that although our study targets Altera FPGAs, the proposed debugging flow is not limited to Altera, and applies equally to FPGAs from other vendors.



(a) Traditional (b) Proposed

Fig. 4. 16-to-1 MUX implementation in 6-input LUTs.

A. An Area-Optimized Multiplexer Implementation

It is well-known that FPGAs are inefficient at implementing multiplexers. Therefore, in this section, a novel multiplexer implementation, optimized in the number of LUTs, is presented. The proposed construction also takes advantage of the bitstream changes (described above).

Fig. 4(a) shows a traditional 16-to-1 MUX implementation in a Stratix III FPGA (the image is a screen capture from Altera's technology map viewer tool). Observe that *five* 6-input LUTs are required. In a traditional MUX, the values of signals on the MUX select inputs can change at any time while the circuit operates. However, in the proposed design process, the selected trace signals do not need to change as the circuit operates. Rather, the set of selected signals is determined by the FPGA bitstream, and as such, may only change between device configurations. This makes an alternative MUX implementation possible – one that consumes only *three* 6-LUTs in the 16-to-1 case.

The new MUX design is based on recognizing that a LUT's internal hardware *contains* a MUX, coupled with SRAM configuration cells. In our design, the LUT's internal MUX forms a portion of the MUX we wish to implement (made possible owing to the MUX select lines being held constant during device operation). Fig. 4(b) shows the proposed 16-to-1 MUX, where the 16 inputs are labeled ($i0$-$i15$). In this case, the LUT configuration SRAM cells (i.e., the truth table) determine which MUX input signal is passed to the output. For the purposes of illustration, in Fig. 4(b), each LUT is labeled with the logic function needed to select the 6^{th} MUX input ($i5$) to the output. Only three LUTs are required: The LUT labeled $f1$ passes input $i5$ to its output. LUT $f2$ can implement any logic function since its output is not observable (however, to save power $f2$ should be programmed to constant logic-`0` or logic-`1`). LUT $f3$ is programmed to pass $f1$ to its output. The proposed design offers significant area savings relative to the traditional design, and allows signal selection via bitstream changes.

B. Debugging with Limited Output Pins

The debugging architecture described above requires multiple output pins if a group of signals is traced in one silicon execution. This approach may not be feasible in cases where the output pins are limited. Therefore, an alternative architecture that utilizes a parallel-in serial-out shift register is presented in Fig. 5.

In Fig. 5, only one output pin is used. Values of the target group are loaded into the shift register in parallel in each clock cycle. Then, the system clock is stopped, and a second *debugging clock*, is used to shift out the stored value. There is a trade-off between the number of output

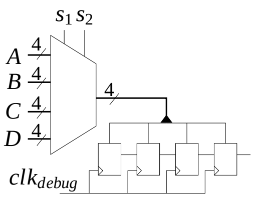


Fig. 5. Multiplexer with a 4-bit Shift Register.

TABLE I
BENCHMARKS.

Ckt.	# ALM	# Reg	Fmax (MHz)	Ckt.	# ALM	# Reg	Fmax (MHz)
ethernet	1323	1256	321.85	main	24483	20046	37.47
mem ctrl	1024	1051	266.95	dfsin	13946	16367	118.29
tmu	2336	3425	168.63	aes	8224	9090	129.10
rsdecoder	658	539	730.46	adpcm	11330	9852	101.58

pins and the test execution time. If more output pins are available, the data can be distributed into multiple shift registers which feed different output pins. This results in fewer clock cycles for retrieving data from the shift registers.

This architecture can be improved to obtain all values stored in the shift registers within one system clock cycle (without stopping the system clock). Instead of shifting the data with a debug clock supplied from off-chip, one can use the on-FPGA PLL to synthesize the debug clock from the system clock, with the debug clock being *n* times faster than the system clock, where *n* is the width of the shift registers. The advantage of this implementation is that the design does not need to be halted after each cycle in order to empty the shift registers. This approach is feasible if the system can be operated at a low frequency.

IV. Experimental Study

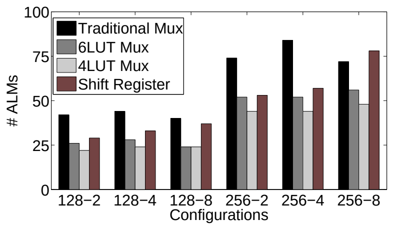
This section presents the area overhead and timing impact of the proposed structures. The structures were integrated into benchmarks selected from the OpenCores and CHStone benchmark suites [13]. The CHStone benchmarks were synthesized from the `C` language to Verilog RTL using a high-level synthesis tool [14]. All RTL benchmarks were then compiled using Altera's *Quartus II 11.0*, targeting the 65*n*m Stratix III FPGA, with a difficult-to-meet timing constraint (1 GHz). Table I summarizes the ALM and register utilization of each original benchmark (i.e. without any debugging structures integrated). The table also shows the post-routing maximum frequency (`Fmax`) of the benchmarks.

In our experimental methodology, registers in each module of each benchmark were randomly selected as tracing candidates. Benchmarks were modified such that traced signals were wired to the top-level of the benchmark and connected to the proposed structures. Altera's synthesis attributes, *keep* and *noprune*, were used to ensure that all signals exist after optimization. In the following discussion, the notation, `m-w`, represents the tracing setting where *m* signals are candidates for tracing and *w* signals are traced concurrently in one silicon execution.

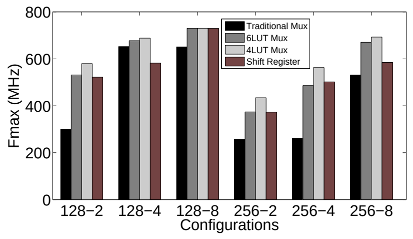
Experimental results for the structures described in Section III are presented in the next subsection, followed by an analysis of the productivity and the stability of the proposed design process.

A. Area Usage and Timing Analysis

The area overhead and `Fmax` of the proposed architectures with various sizes are depicted in Fig. 6. Four implementations are investigated: a traditional MUX implementation, a 6-LUT-based MUX implementation (as proposed in Section III-A), a 4-LUT-based MUX implementation (same as proposed in Section III-A except using 4-LUTs instead of 6-LUTs), and a shift-register-based implementation (as proposed in Section III-B). As shown in Fig. 6(a), the 6-input LUT implementation uses, on average, 35% fewer ALMs than the traditional MUX implementation. The 4-input LUT implementation can further reduce the usage of ALMs. This is because each ALM in a Stratix III device can contain two 4-input LUTs, and Quartus II may merge two 4-input LUTs into one ALM. However, there is no user control to force such an optimization to happen, and therefore, in the remaining experiments, all multiplexers in the proposed structures are implemented with the 6-input LUT approach. For the data in Fig. 6, the shift-register implementation only uses one output pin and is driven with an external debug clock. Due to the presence of the shift register, the area cost is slightly greater than the cost with the full multiplexer implementation.



(a) Area



(b) Fmax

Fig. 6. Area and `Fmax` of multiplexers.

TABLE II
EFFECTS OF AREA-OPTIMIZED MULTIPLEXER.

(a) Area Increase Percentage (ALMs + registers) (%)

Ckt.	128-2	128-4	128-8	256-2	256-4	256-8
ethernet	6.91	7.10	**7.34**	10.87	11.26	**11.53**
mem ctrl	6.12	6.48	**6.90**	10.57	10.66	**11.69**
tmu	5.95	6.02	**6.11**	10.95	10.99	**11.12**
rsdecoder	**11.36**	10.52	9.86	13.03	**19.05**	17.79
main	0.27	0.29	**0.65**	0.77	0.75	**1.15**
dfsin	0.46	**0.52**	0.39	**1.08**	1.06	1.05
aes	1.14	1.31	**1.67**	2.54	2.48	**2.94**
adpcm	1.61	1.52	**1.66**	**1.76**	1.59	1.64

(b) Fmax Change Percentage (%)

Ckt.	128-2	128-4	128-8	256-2	256-4	256-8
ethernet	**-0.28**	-0.02	-0.07	**-0.31**	-0.06	-0.11
mem ctrl	-3.2	**-10.1**	-5.2	-8.19	**-12.15**	-7.23
tmu	1.99	2.12	**2.2**	**1.06**	0.98	0.92
rsdecoder	**-35.06**	-32	-17.51	**-33.99**	-29.87	-28.51
main	-1.81	-1.36	**-4.06**	-0.43	**2.86**	2.16
dfsin	**3.53**	-1.5	3.29	1.57	-0.06	**-3.51**
aes	-0.74	-0.33	**0.77**	0.17	-0.6	**-1.14**
adpcm	**3.62**	2.07	-0.27	**1.79**	-0.5	-0.36

Fig. 6(b) shows the `Fmax` of each MUX implemented in isolation. Since the area-optimized implementation requires fewer ALMs to construct a multiplexer, less parasitic capacitance is introduced on the critical path. Consequently, multiplexers with the 4-input LUT implementation have the highest frequency in most cases.

Table II(a) reports the percentage increase in ALMs and registers of benchmarks when the area-optimized multiplexer is integrated. Two groups of tracing settings are considered. The worst-case in each group is shown in bold. Results show that in most cases the area overhead is less than 10%. The `Fmax` of the benchmarks with the same tracing settings is reported in Table II(b). Overall, `Fmax` is not affected greatly – changes are mainly due to algorithmic noise. The only exception is with `rsdecoder`, with reason being that the critical path for this benchmark is altered to pass through the multiplexer.

Similar to Table II(a) and Table II(b), the effect of the shift register-based structure on the area and `Fmax` of benchmarks is summarized in Table III(a) and Table III(b), respectively. Here, instead of using an external debug clock, a faster debug clock is generated from the system clock using the Stratix III PLL. The faster clock allows us to shift out the content of the shift-register within one system clock cycle. As expected, because of the additional shift registers, the overall area overhead can be a bit higher than the area overhead of the full multiplexer discussed previously. Furthermore, `Fmax` drops significantly in all cases – the system clock speed is limited by the debug clock speed. For three of the eight benchmarks, `Fmax` drops more than 50%.

B. Productivity and Stability

In the last set of the experiments, we evaluate the productivity and stability of the conventional design process. Altera's SignalTap

TABLE III
EFFECTS OF AREA-OPTIMIZED MULTIPLEXERS WITH SHIFT REGISTERS.

(a) Area Increase Percentage (ALMs + registers) (%)

Ckt.	128-2	128-4	128-8	256-2	256-4	256-8
ethernet	6.81	**7.43**	7.16	9.71	10.53	**11.42**
mem ctrl	6.72	6.57	**6.92**	10.56	11.11	**11.93**
tmu	6.56	6.20	**6.70**	9.81	10.29	**10.76**
rsdecoder	12.36	**13.37**	12.53	19.21	**20.21**	19.80
main	0.21	**0.25**	**0.25**	0.60	**0.64**	0.63
dfsin	0.29	0.27	**0.39**	0.73	0.76	**0.85**
aes	1.09	1.02	**1.13**	2.14	**2.24**	2.17
adpcm	1.22	1.20	**1.49**	1.70	1.77	**1.88**

(b) Fmax Change Percentage (%)

Ckt.	128-2	128-4	128-8	256-2	256-4	256-8
ethernet	-42.76	-44.2	**-44.89**	-51.1	**-54.26**	-53.32
mem ctrl	**-29.25**	-28.05	-28.67	**-39.87**	-35.37	-31.33
tmu	-5.24	**-6.08**	-6.03	-0.43	**-9.68**	-7.14
rsdecoder	**-75.55**	-71.96	-69.02	-75	-76.25	**-76.41**
main	**0.77**	0.61	-0.72	**-2.86**	-1.49	0.43
dfsin	**-10.74**	-9.38	-8.57	**-10.24**	-7.75	-6.02
aes	**-4.93**	-2.98	-1.9	-11.11	-10.22	**-11.96**
adpcm	**-3.63**	1.1	1.55	**4.1**	3.36	2.63

TABLE IV
COMPILATION TIME OF SIGNALTAP.

Ckt.	128-8				256-8			
	Prop. (sec)	SignalTap (sec)			Prop. (sec)	SignalTap (sec)		
		First	Incr.	Total		First	Incr.	Total
ethernet	139	134	117	2006	141	134	119	3946
mem ctrl	150	143	124	2129	156	143	123	4073
tmu	169	161	137	2354	179	161	140	4639
rsdecoder	106	103	99	1685	109	103	98	3233
main	1449	1448	293	6141	1453	1448	290	10737
dfsin	706	696	216	4150	711	696	217	7648
aes	465	453	186	3428	466	453	184	5901
adpcm	634	615	226	4234	639	615	225	7815

is used as the embedded logic analyzer. As mentioned in Section III, due to the size of SignalTap, acquiring trace data for a large number of signals is often achieved by successively tracing multiple smaller groups. Recompilation is required when different signals are selected.

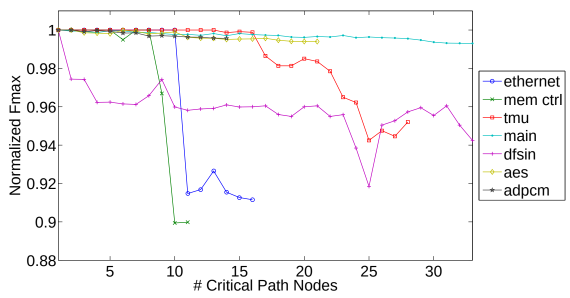
The experiment is carried out as follows. Two tracing settings are studied: `128-8` and `256-8`. In order to use the incremental compilation feature in Quartus II, only post-fitting signals are considered. First, the design is compiled without the SignalTap module. 128(256) post-fitting nodes are randomly selected after the first compilation. Next, eight signals from the set are monitored. The procedure is repeated until all 128(256) signals are traced.

The compilation time results are summarized in Table IV. The first column lists the benchmarks. The next four columns report the results for the first tracing setting: the compilation time of the proposed process, the first compilation of the SignalTap process, the average compilation time of each data acquisition session and the total cumulative compilation time of the SignalTap-based debugging process. The result of the second tracing setting is reported in the final four columns. As shown in the table, since the proposed bitstream-modifications-only process only requires one compilation, the compilation time roughly equals to the first compilation of the SignalTap process. Although incremental compilation reduces the compilation time by 4%-80%, each additional compilation adds time overhead. Overall, the proposed process can save up to 93% (i.e., 139/2006 for `ethernet`) in the case of the `128-8` scenario, and 97% (i.e., 109/3233 for `rsdecoder`) in the case of `256-8`.

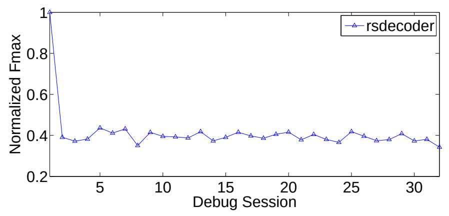
Incremental compilation tries to preserve the engineering effort from a previous compilation to minimize the impact to design performance. While it does well in many cases, experiments show that `Fmax` can still vary when the monitored signals are on the critical path. Performance stability results are shown in Fig. 7(a). In each case, a total of 32 signals are traced. The x-axis of the plot is the number of traced signals that are on the critical path. The y-axis is the normalized `Fmax`, where the base is the `Fmax` of the original benchmark. One can see that `Fmax` drops in various degrees, as much as 10%. It all depends on what signals are monitored.

For designs that can be operated at a very high frequency, the SignalTap module can in fact be where the critical path resides. In this case, monitoring *any* set of signals can change `Fmax`, as shown in Fig. 7(b). The x-axis of the plot is the data acquisition session, where 8 signals are traced in each session with 32 sessions in total. The plot shows that `Fmax` is unstable from one session to another.

In the proposed debugging approach, since only a single execution



(a) Tracing nodes on the critical path



(b) Tracing random nodes

Fig. 7. Stability of SignalTap.

of synthesis, place and route are needed, circuit speed performance is stable as different sets of signals are traced.

V. CONCLUSIONS AND FUTURE WORK

Functional debugging using FPGA devices provides several advantages over the traditional software simulation approach. This work presents a set of hardware structures to take the advantage of the FPGA reconfigurability feature to enhance the observability for debugging. Furthermore, experimental results demonstrate that the new techniques can improve the productivity of the debugging process up to 30×. One of the extensions to this work can be the integration of debug features, such as trigger events, to the proposed structures to enhance the debugging ability. Another interesting extension is developing a debugging algorithm that utilizes the proposed structures to provide an efficient FPGA debugging environment. Lastly, the proposed hardware and methods can be used in conjunction with FPGA embedded SRAM blocks to store trace data across multiple execution cycles before sending the data to the chip output pin(s).

REFERENCES

[1] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. on CAD*, vol. 26, no. 2, pp. 203–215, 2007.
[2] M. Gort and J. Anderson, "Deterministic multi-core parallel routing for FPGAs," in *IEEE FPL*, 2010, pp. 78 –86.
[3] L. Lagadec and D. Picard, "Software-like debugging methodology for reconfigurable platforms," in *IEEE Int'l Symp. on Parallel and Distributed Processing*, 2009, pp. 1–4.
[4] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting bitstreams for debugging FPGA circuits," in *IEEE FCCM*, 2001, pp. 41–50.
[5] *ChipScope ILA Tools Tutorial*, Xilinx Inc., San Jose, CA, 2003.
[6] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *IEEE DAC*, 2006, pp. 7–12.
[7] B. Quinton and S. Wilton, "Programmable logic core based post-silicon debug for SoCs," in *IEEE Int'l Silicon Debug and Diagnosis Workshop*, 2007.
[8] H. F. Ko and N. Nicolici, "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug," *IEEE Trans. on CAD*, vol. 28, no. 2, pp. 285 – 297, 2009.
[9] Y.-S. Yang, N. Nicolici, and A. Veneris, "Automating data analysis and acquisition setup in a silicon debug environment," *IEEE Trans. on VLSI*, 2011.
[10] E. Hung and S. Wilton, "Speculative debug insertion for FPGAs," in *IEEE FPL*, 2011, pp. 524–531.
[11] *Design Debugging Using the SignapTap II Logic Analyzer*, Altera, Corp., San Jose, CA, 2011.
[12] *Increasing Productivity With Quartus II Incremental Compilation*, Altera Corp., San Jose, CA, 2008.
[13] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.
[14] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *ACM FPGA*, 2011, pp. 33–36.