Fast Turnaround HLS Debugging using Dependency Analysis and Debug Overlays

AL-SHAHNA JAMAL, University of British Columbia, Canada
ELI CAHILL, Brigham Young University, USA
JEFFREY GOEDERS, Brigham Young University, USA
STEVEN J.E. WILTON, University of British Columbia, Canada

High-level synthesis (HLS) has gained considerable traction over the recent years as it allows for faster development and verification of hardware accelerators than traditional RTL design. While HLS allows for most bugs to be caught during software verification, certain non-deterministic or data-dependent bugs still require debugging the actual hardware system during execution. Recent work has focused on techniques to allow designers to perform in-system debug of HLS circuits in the context of the original software code; however, like RTL debug, the user must still determine the root-cause of a bug using small execution traces, with lengthy debug turns. In this work we demonstrate techniques aimed at reducing the time HLS designers spend performing in-system debug. Our approaches consist of performing data dependency analysis in order to guide the user in selecting which variables are observed by the debug instrumentation, as well as an associated debug overlay that allows for rapid reconfiguration of the debug logic, enabling rapid switching of variable observation between debug iterations. In addition, our overlay provides additional debug capability, such as selective function tracing and conditional buffer freeze points. We explore the area overhead of these different overlay features, showing a basic overlay with only a 1.7% increase in area overhead from the baseline debug instrumentation, while a deluxe variant offers 2x-7x improvement in trace buffer memory utilization with conditional buffer freeze support.

ACM Reference Format:
Al-Shahna Jamal, Eli Cahill, Jeffrey Goeders, and Steven J.E. Wilton. 2019. Fast Turnaround HLS Debugging using Dependency Analysis and Debug Overlays. *ACM Trans. Reconfig. Technol. Syst.* 12, 0, Article 0 (2019), 26 pages. https://doi.org/00000000/00000000

1 INTRODUCTION

Recent years have seen the emergence of Field-Programmable Gate Arrays (FPGAs) as mainstream compute accelerators. Companies such as Amazon, IBM, Baidu, and Microsoft have invested significant resources understanding how FPGAs can be used to accelerate cloud-based computing. For FPGAs to thrive in this new role, new programming frameworks are essential. FPGA vendors have responded by creating high-level synthesis (HLS) tools [2, 23] which allow designers to specify behaviour in a software language (C/OpenCL) and automatically compile this code to hardware.

Authors' addresses: Al-Shahna Jamal, alshahnaj@ece.ubc.ca, University of British Columbia, Department of Electrical and Computer Engineering, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4; Eli Cahill, ecahill42@gmail.com, Brigham Young University, 450 EB, BYU, Provo, UT, USA, 84602; Jeffrey Goeders, jgoeders@byu.edu, Brigham Young University, 450 EB, BYU, Provo, UT, USA, 84602; Steven J.E. Wilton, stevew@ece.ubc.ca, University of British Columbia, Department of Electrical and Computer Engineering, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1936-7406/2019/0-ART0 $15.00
https://doi.org/00000000/00000000

HLS technology promises significant productivity improvements and may someday open the door for software designers to enjoy the speed and energy advantages of FPGAs [7, 19].

In order to take advantage of HLS technology, techniques for debugging these designs are essential. Many bugs can be identified by executing the source C code on a workstation and using a software-oriented debugging tool. However, some bugs can only be observed when executing the design on-chip. Executing a design on-chip allows for much longer execution paths and provides the ability to understand how a design works as part of a larger system, with real input traffic.

Debugging a running FPGA is challenging. Due to limited I/O pins, it is necessary to store data on-chip using an embedded logic analyzer to capture short recordings of select signals into on-chip memories. These recordings can be reviewed after the chip has run to help the user understand the operation of their design to narrow down the root cause of unexpected behaviour. A second challenge that occurs in the context of an HLS design is to understand how the hardware signals from the machine-generated circuit correspond to the original software code. This is made even more difficult in the presence of heavy compiler optimizations.

Recent work [4, 13, 14] presents techniques to extract software-to-hardware mapping information from the HLS tool, allowing the designer to debug in the context of the original source code. The designer is provided a software-like debug environment where they can insert breakpoints, step through code, and inspect variables. As with RTL-level techniques [2, 23], debugging is done using captured execution traces obtained from the on-chip trace buffers; the work in [13] shows how scheduling information from the HLS tool can be used to compress the trace information on a circuit-by-circuit basis to maximize the amount of data stored in the trace buffer.

Due to limited on-chip memory and/or slow bandwidth to off-chip memory, the user can typically capture data corresponding to only milliseconds of execution. To maximize the usefulness of the trace buffer data, a designer would likely trace only a few “important” signals during a single debug run. Typically, several debug iterations will be required for the user to pin-point the cause of a bug. If subsequent debug iterations require changing the signals to be traced, the design must be re-instrumented and recompiled. Compilation can be slow and can take several hours or even days for large designs [21]. This significantly limits debug productivity.

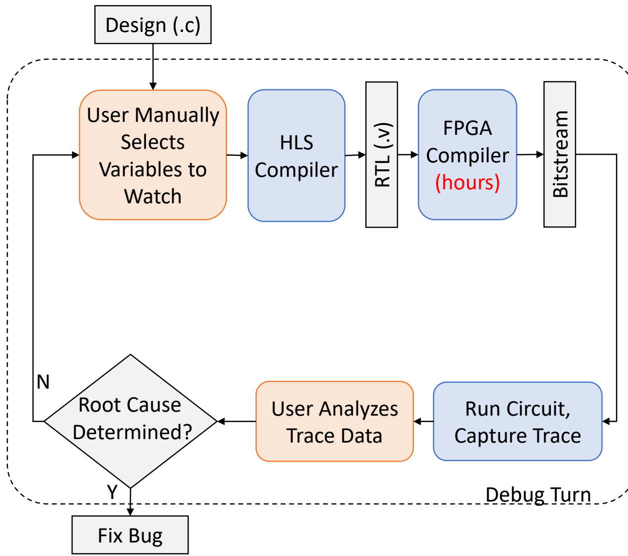
In this work, we present two techniques to accelerate HLS debugging. First we describe an *HLS-specific Debug Overlay*, which allows the user to alter which variables are recorded between debug iterations, without necessitating a full place and route recompile of the hardware. This provides *software-like debug turn-around times for circuits created by an HLS tool* (i.e. reconfiguration on the order of hundreds of milliseconds). Second, we describe *Dependency Driven Debug* in which data dependency analysis techniques are used to drive variable selection techniques for HLS debug and to ease the user analysis of the captured trace data. We show how these two techniques can be combined to significantly accelerate debug.

This work builds on our previous conference paper which describes the overlay architecture [16]. Compared to that paper, we have added dependency-driven signal selection and show how it can be used in concert with the overlay.

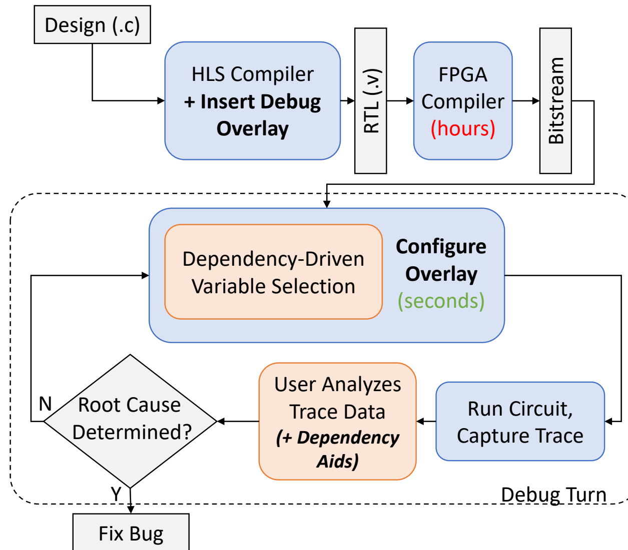
This paper is organized as follows. The baseline flow is presented in Section 2. The modified flow that incorporates our two techniques is then described in Section 3. Sections 4 and 5 provide more details on the overlay and dependency analysis respectively, and results are presented in Section 6.

2 BASELINE DEBUG FLOW AND INSTRUMENTATION

To make our work concrete, we demonstrate our techniques on top of a previously published HLS debug flow [13]. This baseline flow is shown in Figure 1a. In this flow, a C program is compiled to a Hardware Description Language (HDL) representation (we use LegUp [6]). The HLS tool is modified to automatically add signal-tracing instrumentation to the user circuit, and creates a



(a) Previous Work (slow HLS debug)



(b) This Work (rapid HLS debug)

Fig. 1. The debugging flow presented in this work. The lengthy FPGA compile is moved outside of the debug loop, allowing for faster debugging iterations. This is made possible through a debug overlay that can alter observed variables at runtime. This overlay works in conjunction with dependency analysis to aid the user in the debug process.

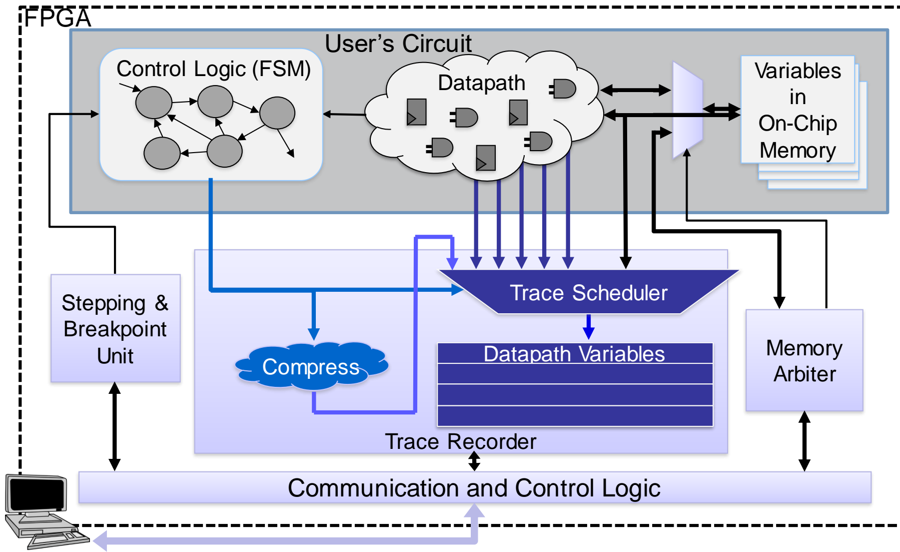


Fig. 2. Baseline Instrumentation from [13] shows interaction with User Circuit and Debug Workstation

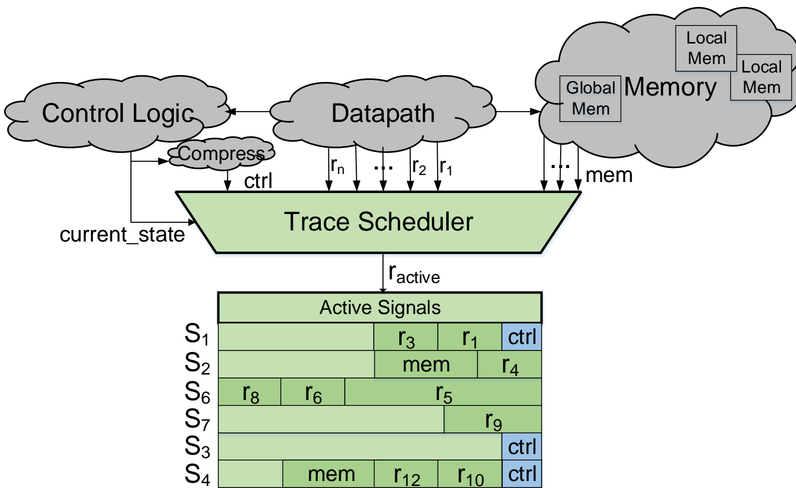


Fig. 3. Baseline Trace Scheduler Instrumentation and trace buffer contents of hypothetical program

debug database, which contains a mapping between C-code variables and signals/memories in the RTL code, as well as scheduling information regarding when variables are updated. The circuit is then compiled using a vendor-specific tool-chain (we use Quartus II) and implemented on an FPGA.

Figure 2 shows the instrumentation from [13]. We assume that there is a single trace buffer, which is used to store the history of user-visible variables, as well as sufficient control-flow information such that the control path can be reconstructed by the offline debug GUI. User-visible variables can either correspond to signals in the user circuit datapath or to global or local memories. Variables

corresponding to signals in the design can be recorded directly, while variables in memory are captured by recording the *address* and *data* signals when a write operation occurs. The HLS schedule is analyzed to construct the *Trace Scheduler* logic, which dictates which signals are recorded in the trace buffer each cycle. Figure 3 shows a detailed view of the cycle-by-cycle signal tracing technique, and a portion of a hypothetical debug trace. In the first cycle (State S_1), user variables r_3 and r_1 are updated, so their values are stored in the first line of the trace buffer. Control flow information is also stored in this cycle. In the second cycle (State S_2), user variable r_4 is updated, along with a variable that has been mapped to the global memory, so both updates are stored in the trace buffer. Notice that the width of the trace buffer must at least match the number of bits to be written in the worst-case state.

Most relevant to this work, is recognizing that this trace scheduler must be constructed on a circuit-by-circuit basis when the user circuit is instrumented as this circuitry is optimized for both the user circuit and the set of variables that the user has elected to record. Reference [13] shows that this circuit-specific tracing approach leads to an improvement in the trace window size of up to 127x.

As the FPGA runs, the instrumentation records the behaviour of identified signals in the on-chip trace buffer. After the run is complete, the user can read this data, and use it in conjunction with a software-like debugging user interface to help pin-point the root cause of the behaviour. If the root cause can not be identified, the designer can change the traced signals, recompile the design, and repeat. Often, many debug iterations will be required before the designer can identify the cause of the bug.

3 NEW DEBUG FLOW

Our enhanced flow is shown in Figure 1b. Like the baseline flow, the HLS tool is modified to insert instrumentation. Unlike the baseline flow, this instrumentation is in the form of a flexible *overlay* that can be configured to implement different debug scenarios between iterations without requiring a recompilation. Debug scenarios may describe specific variables that should be captured or regions of code that should be traced. At debug time, between debug turns, the user can configure the overlay to implement a specific debug scenario *without recompiling the design or overlay*. Our overlay architecture will be described in Section 4. Central to the overlay is the ability for the user to modify the set of variables that are being recorded in the trace buffer without recompiling the design. This allows the user to modify which parts of the code is being traced, and also allows the user to trade off the number of variables recorded with the trace window depth. Initially, the user may trace many variables to get a quick idea of the overall state of the system. In later debug iterations, the user may decide to focus on only a small subset of variables that are believed to be responsible for the problem.

Once the instrumentation has been inserted, the circuit and instrumentation are compiled as normal. The flexibility of the overlay leads to increased area and delay overhead compared to the fixed instrumentation used in the baseline; this overhead will be evaluated in Section 6.

Before running the circuit, the user configures the overlay to specify which variables are to be traced. In our flow, this selection is guided by automatic dependency analysis. With knowledge of which output variables are important (perhaps those that exhibit the incorrect behaviour), the automatic dependency analysis works "backwards" through the design, identifying variables that affect the output. These are the variables that are likely to provide clues to the root cause of the error, and thus are important to trace. Section 5 discusses our approach to perform automatic dependency analysis in an HLS tool (LegUp), and includes experimental results that show the feasibility of applying such an approach.

Static | Recorded

Line	Var	IR Instruction	Function	Line Number
--Globals--				
g arr			--Global--	5
fun_arr[i] = fun_arr[i] * x;	g_arr, loc_arr	%3 = load i32* %scevgep, al...	fun	11
fun_arr[i] = fun_arr[i] * x;	g_arr, loc_arr	store i32 %4, i32* %scevgep...	fun	11
g_arr[i] = i + i;	g_arr	store i32 %3, i32* %scevgep...	main	20
fun				
fun arr			fun	8
x			fun	8
ret val			fun	9
ret_val += fun_arr[i];	ret_val	%5 = add nsw i32 %4, %ret_...	fun	12
i			fun	10
for (int i = 0; i < SIZE; i++){	i	%6 = add nsw i32 %2, 1, !db...	fun	10
main				
loc arr			main	18
fun_arr[i] = fun_arr[i] * x;	g_arr, loc_arr	%3 = load i32* %scevgep, al...	fun	11
fun_arr[i] = fun_arr[i] * x;	g_arr, loc_arr	store i32 %4, i32* %scevgep...	fun	11
loc_arr[i] = i * i;	loc_arr	store i32 %4, i32* %scevgep...	main	21
i			main	19
for (int i = 0; i < SIZE; i++){	i	%5 = add nsw i32 %2, 1, !db...	main	19
y			main	23
int y = fun(g_arr, 20);	y	%7 = tail call fastcc i32 @fu...	main	23
z			main	24
int z = fun(loc_arr, y);	z	%9 = call fastcc i32 @fun(i3...	main	24
int y = fun(g_arr, 20);	y	%7 = tail call fastcc i32 @fu...	main	23
return ret_val;	[unresolved]	ret i32 %5, !dbg !49	fun	14
ret_val += fun_arr[i];	ret_val	%5 = add nsw i32 %4, %ret_...	fun	12
fun_arr[i] = fun_arr[i] * x;	g_arr, loc_arr	%3 = load i32* %scevgep, al...	fun	11
int fun(int* fun_arr, int x){	x		fun	8
int y = fun(g_arr, 20);	y	%7 = tail call fastcc i32 @fu...	main	23
return ret_val;	[unresolved]	ret i32 %5, !dbg !49	fun	14
ret_val += fun_arr[i];	ret_val	%5 = add nsw i32 %4, %ret_...	fun	12

Fig. 4. Dependency information in our prototype debugger

Once the user has run the circuit and gathered trace data, he or she can use this with a debug user interface as in the baseline flow. The execution trace collected may represent tens of thousands of cycles of execution [13], so sifting through this data may be time-consuming. To address this, in our enhanced flow, the debug user interface is enhanced with the ability to step back through dependencies, using the same dependency engine as is used for overlay configuration. Figure 4 shows a sample of how this dependency information is presented to the user; for each variable within a function, we provide a list of C-code instructions that it depends upon. Each variable can be further expanded to explore dependencies further back in the program.

As in the baseline flow, this iterative debug process is continued until the designer identifies the root cause of the bug. After debugging is complete, the user may choose to remove the overlay for production (possibly allowing the user to target a smaller FPGA) or leave the overlay in place, but disable it.

4 OVERLAY ARCHITECTURE

This section describes our overlay architecture family which makes it possible to rapidly reconfigure the debug instrumentation. The key technical challenge in our approach is creating an architecture for the instrumentation overlay that is flexible enough to support the types of debug capabilities that we expect to be useful, yet has small area overhead and has as little impact on the clock frequency of the user circuit as possible.

4.1 Related Work

Unlike previous debug overlays [8], our architecture is optimized for HLS circuits and is intended to be tightly integrated into an HLS tool.

Work by Yang [24, 25], Fezzardi [9, 10], and Campbell [5] has focused on automatically locating the cause of bugs in HLS code. While locating bugs automatically is far preferable to having a user perform debugging, it relies on detecting discrepancies between the hardware and software, and is thus limited to only certain types of bugs that cause this behavior, such as uninitialized values, out of bounds errors, HLS engine bugs, or interface errors. In contrast, this work, like those in [3, 4, 12, 13, 17, 18], can be used to locate general functional bugs, by providing the user with observability into the design. While the work in this paper aids the user by making it faster to locate the root cause of behaviors, it does not perform automatic bug detection.

In [11], Fezzardi et al. present techniques to efficiently compress control-flow traces of HLS programs. This is similar in objective to the work discussed previously from [13], which is to make efficient use of limited on-chip tracing resources. While both of these works compile an optimized tracing network into the user's design, this work focuses on inserting a *flexible* tracing network that can be dynamically modified to avoid recompile time. While we build our architecture upon the work in [13], it's possible similar techniques could be used to add runtime reconfiguration to other tracing networks, such as the one presented in [11].

4.2 Overlay Capabilities

Our overlay is designed to support a suite of useful run-time debug capabilities; the primary capability is run-time reconfiguration of which signals are captured, thus enabling the dependency-driven debug flow discussed previously. However, we additionally wanted to explore whether we could provide other debug capabilities that would be helpful to an HLS designer. To quantify the flexibility of the debug overlay, we define three "capabilities" that the instrumentation can provide: *Selective Variable Tracing*, *Selective Function Tracing*, and *Conditional Buffer Freeze*. The overlay we present in this paper can be configured to provide any or all of these capabilities.

- **Selective Variable Tracing** This refers to the capability to configure the overlay, at debug time, to specify which variables should be traced. In [13], signals corresponding to *all* user-visible variables in the source code are traced. This provides the most software-like debug experience, since it allows the user to trace through a recorded execution and display the value of every variable at every step. However, as described earlier, for large designs, and in particular designs with many parallel functional units, recording the behaviour of all variables may quickly consume the limited on-chip trace buffer. If fewer variables are recorded, then a longer *trace window* can be stored in the trace buffer, possibly making it easier for the user to understand the behaviour of the design and deduce the root cause of a bug. In our implementation, the overlay is flexible enough that the user can specify any number of variables to trace (including all of them) within a debug turn.
- **Selective Function Tracing** This refers to the capability to specify, at debug time, specific source code function(s) of interest. Once the user identifies specific functions, the fabric records the activity only within those functions. This capability may be useful if the user has narrowed down the cause of a bug to a specific erroneous function output. By recording only data within the function, a longer trace of the function behaviour can be obtained.
- **Conditional Buffer Freeze** Conditional Buffer Freeze refers to the capability to specify, at debug time, a condition that, when true, causes recording of data in the trace buffer to halt. After recording halts, the user can read out information in the trace buffer to understand what led up to that point. As an example, the user may set a freeze point to occur when a particular error flag goes high, or an argument to a function is not within an expected range. We distinguish between a freeze point and a breakpoint; in the former, the execution

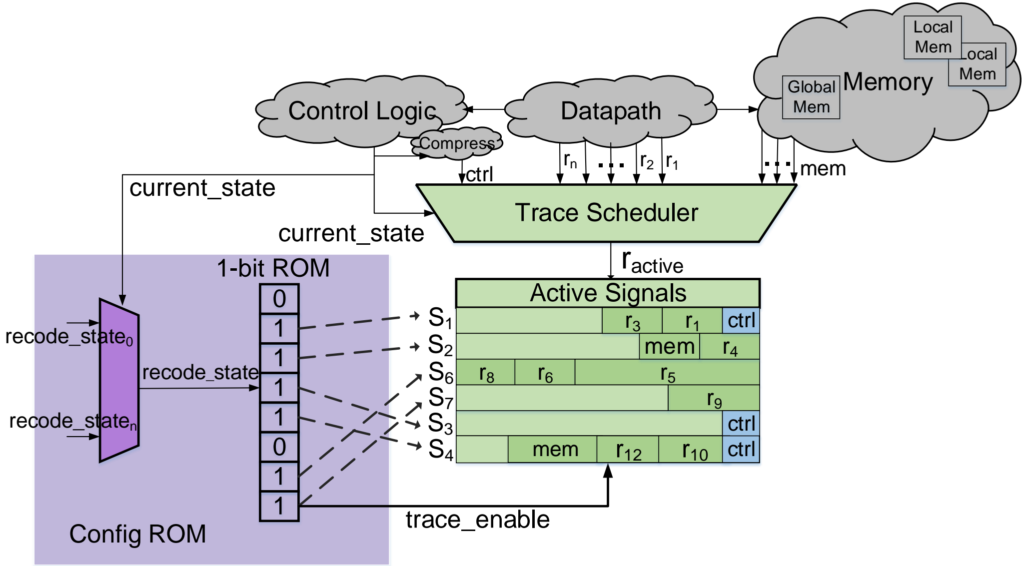


Fig. 5. Selective Variable Tracing - Variant A

of the chip may continue, however, by freezing the contents of the trace buffer, the execution history up until the freeze point is preserved.

In the next subsections, we describe how each of these capabilities are supported in our architecture.

4.3 Enabling Selective Variable Tracing

We first describe the architectural enhancements that will allow the user to select, at run time, a subset of variables to trace. We present two variants of the architecture: *Variant A* which has the least overhead but may not make effective use of the trace buffer, and *Variant B* which will make more efficient use of the trace buffer, but at the expense of more area overhead.

4.3.1 Selective Variable Tracing Architecture - Variant A. The first variant we consider is shown in Figure 5. The Signal Trace Scheduler and the Trace Buffer (both shown in green) are the same as in the baseline architecture (Figure 3). The 1-bit wide Config ROM and the multiplexer that feeds the address lines (shown on the bottom left) are new. The Config ROM contains the overlay configuration bits that encode which user variable(s) are to be recorded. Intuitively, it would make sense to include one configuration bit for each variable in the user circuit (to indicate if that variable should be recorded or not recorded), however, we found that this approach leads to large decoding logic which unacceptably increases the overhead of our overlay. Instead, we associate each configuration bit with one *state* in the user circuit. If a user variable is to be recorded, the configuration bits corresponding to all states in which that variable is updated are set to 1; this enables the write enable line of the trace buffer during those states. Since we wish to record all control flow information regardless of the user's variable selection, configuration bits corresponding to states in which control flow information is to be stored are also set to 1. As an example, Figure 6(a) shows the trace buffer contents after executing a hypothetical program in which all user variables are traced. If the user wishes to record only $r1$ and $r9$, then the trace buffer is *not* updated during states $S2$ and $S6$, leading to the more efficient packing in Figure 6(b) (note that the buffer still needs

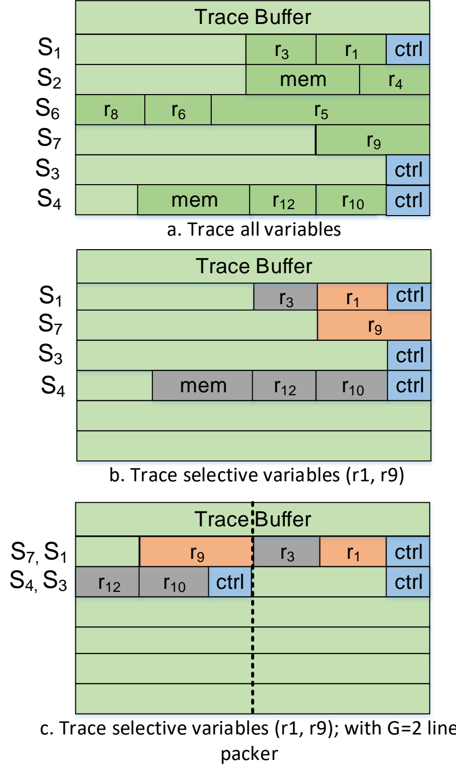


Fig. 6. Selective Variable Tracing Examples

to be updated in states $S3$ and $S4$ because control-flow information is written during those states). This strategy causes the trace buffer to fill more slowly, meaning at the end of the run, a larger window of execution is available in the trace buffer, providing more information for the user as he or she seeks the root cause of a bug.

Note that, since several variables may be updated in the same state, unselected variables may also be inadvertently recorded. In Figure 6(b), even though the user has not selected $r3$, the value in $r3$ is still recorded in the first state, since $r1$ has been selected. Similarly, $r12$ and $r10$ are recorded in the fourth line since the control flow information in that state must be recorded. This leads to a slightly less efficient use of trace buffer space than approaches such as [3] in which the compression circuitry is optimized to store *only* those variables that have been selected; in Section 6 we will quantify this impact.

In a naïve implementation, the depth of the 1-bit wide ROM would be equal to the number of states in the user circuit. However, in some states, no user variables are updated, and no control flow data needs to be captured, so no configuration bit is necessary. Thus, we use a multiplexer (the left-most multiplexer in Figure 5) to recode the state number (which is obtained from the user circuit) to a linear sequence of states in which at least one variable is updated (we call these *trace states*). The recoded state is then used to address the Config ROM to acquire the corresponding configuration bit for the trace buffer. In our benchmark circuits we found that this approach reduces the depth of the ROM by approximately 55%. More importantly, it provides compatibility with HLS tools that do not encode their states sequentially. Additionally, since the multiplexer inputs are

constant, significant area is reclaimed by the logic synthesis algorithm in the FPGA CAD suite as it optimizes the circuit, and this multiplexer requires little additional logic.

In our implementation, the 1-bit ROM is implemented using one or more embedded FPGA memory blocks. This allows us to change the configuration bits at debug time, to implement a new debug scenario, without recompiling the circuit.

At debug time, an algorithm is needed to map a debug scenario (specified by the user using a GUI) to the values that will be stored in the overlay configuration cells. In the architecture of Figure 5, each overlay configuration bit corresponds to one state in the schedule of the user circuit. When the HLS tool creates the user circuit, it also creates a debug database which contains a list of all user variables that are updated in each state, as well as a list of when control flow information is generated. At debug time, when the user selects signals to observe, it is then straightforward to use this debug database to set the overlay configuration bits appropriately.

4.3.2 Selective Variable Tracing Architecture - Variant B. When the overlay is constructed (at compile time), the trace buffer width is set to be equal to the largest number of bits that must be stored in a single state. In the example of Figure 6(a), the worst-case state is $S6$, so the width of the trace buffer is set to be the sum of the widths of all variables to be recorded in $S6$. At debug time, the trace buffer width can not be changed, meaning we may waste space in the trace buffer. In Figure 6(b), the trace buffer is not updated in $S6$, meaning the upper-order bits of every line in trace buffer remain unused. The severity of the problem increases as we reduce the number of variables selected. In this subsection, we show an alternative architecture, *Variant B*, which addresses this concern.

The major difference from the previous variant is the introduction of a *line packer* module, which is designed to pack partially used lines of trace data together into a single line in the trace buffer, thus making better use of memory. Figure 7 illustrates this module, and shows how data from two different states ($S1$ and $S7$, followed by $S3$ and $S4$) are combined into a single line in the trace buffer.

The line packer works by breaking up each incoming data line into G equally sized words (G is an architectural parameter representing the granularity of the line packer). The example in Figure 7 shows a scenario where G=2, whereas Figure 8 shows the detailed architecture for a line packer with G=4. Once the words have been split, a series of multiplexers will steer as many words of data into the current line of the trace buffer as can be accommodated, while the remaining words are temporarily stored in overflow registers, and steered into appropriate locations in the trace buffer the following cycle.

Increasing G splits the incoming data into smaller words, allowing for a more fine grained packing, saving memory. However, as we will show in the results in Section 6, increasing G also increases the area of the line packer.

In order for the line packer to operate, it must know which of the incoming G words contain important data to save to the buffer, and which contain data that can be discarded. To accomplish this, the 1-bit Config ROM from Variant A, is replaced with a multibit ROM. While in Variant A a single bit indicates whether or not the data line for a given state would be recorded, in Variant B, the ROM indicates how many words of the data line should be saved. Thus, if the data line is split into G words, the width of the Config ROM must be $\lceil log_2(G+1)\rceil$ bits.

It should be noted, that for a value n retrieved from the Config ROM, the line packer will save the *lower contiguous n* words of the data line. For example, in Figure 7, this means that if the user did not choose to observe the *mem* update in $S4$, then only the first word is saved. If *mem* was selected, then the entire line would be saved regardless of what the user chose to observe in the lower word. Although it would be possible to design a line packer that saved arbitrary (not necessarily

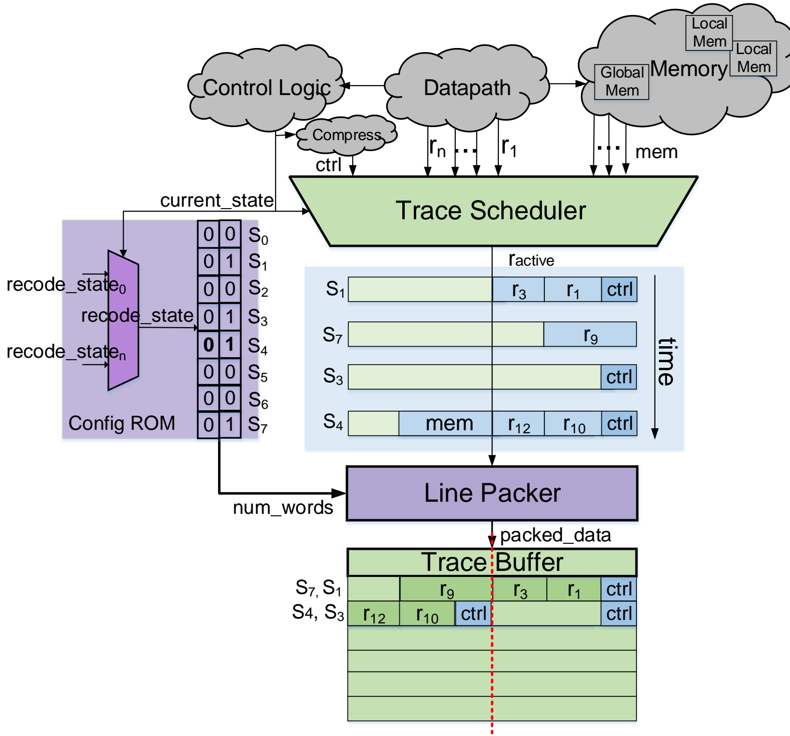


Fig. 7. Selective Variable Selection - Variant B

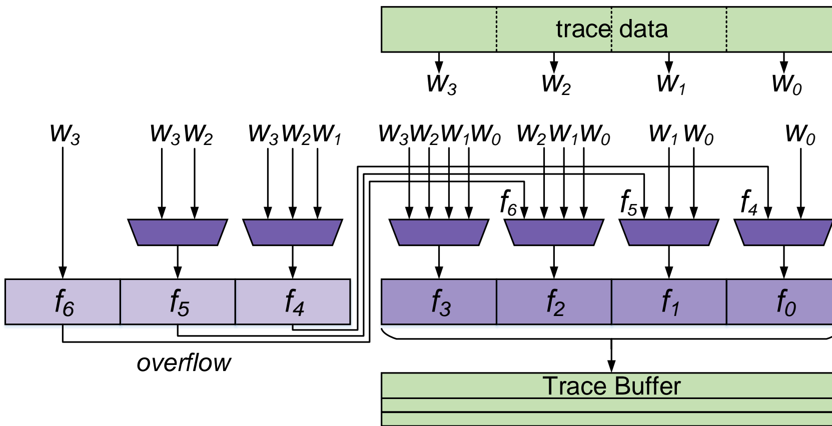


Fig. 8. Line packer architecture with G=4. Incoming trace data is split into G words ($w_{3..0}$), with the config ROM indicating how many incoming words contain valid data. The words are accumulated, right-to-left, in the $2G-1$ data registers ($f_{6..0}$) until the lower G words ($f_{3..0}$) are occupied. When this occurs, a full trace line is written out to the buffer, and the registers are shifted to the right by G.

contiguous) words from the data line, we found such a module would require a prohibitively large amount of area to build.

Like Variant A, Variant B is accompanied by a mapping algorithm that maps the user's variable selection to the ROM words. In this case, the algorithm can use the HLS schedule to determine how many selected variables are written each cycle and the position of these variables in the trace buffer output, and can set the ROM bits accordingly.

4.4 Enabling Selective Function Tracing

To enable selective function tracing, no additional changes to the architecture are required, beyond those described in the last section for selective variable tracing. Changes are required in the CAD algorithm that maps the debug scenario to the overlay configuration bits. When the user selects one or more functions to be traced, the algorithm uses the debug database to determine which user variable assignments are associated with each selected function, and which states correspond to these assignments.

A complexity arises if the HLS tool makes extensive use of function inlining. In such cases, it is often difficult to crisply delineate which state(s) correspond to the inlined function, and which correspond to the parent function, since operations from each function can be mapped to the same state. To accommodate inlining, if an inlined function is selected for tracing, we conservatively trace all states in the parent function as well.

4.5 Enabling Conditional Buffer Freeze

Our conditional buffer freeze architecture is parametrized by C which is the number of conditions upon which a freeze point can depend. As shown in Figure 9, the architecture consists of C comparison subunits and a single Trace Buffer Write Controller. Each comparison subunit monitors the running circuit for a single condition, and the supported operations of the comparator are $=$, $>$, $<$, $\geq$, $\leq$, and $\neq$. Each subunit contains a wide Configuration Register that can be configured at debug time via the debug UART port connected to the GUI.

It should be noted that our freeze point architecture triggers when a specific *assignment* assigns a value that meets a specified condition. In the baseline architecture, the value of each assignment appears at the output of the trace scheduler multiplexer during a specific state. When the user wishes to set a conditional freeze point, the debug database is used to determine the state that the value from this assignment will appear at the output of the trace multiplexer, and this state is stored in the *state* field in the Configuration Register. Since several variable values may appear at the output of the trace multiplexer in the same cycle, it is necessary to select the proper subset of bits in the multi-bit trace multiplexer output. Rather than providing a barrel shifter (which would be large), a *data_mask* is generated (by the algorithm that maps the debug scenario to the overlay) to isolate the variable selected, along with a *target_value* which has been scaled appropriately so that the comparison can be performed without shifting.

The Trace Buffer Write Controller receives signals from all C comparison subunits, and when any of these signals becomes true, it stops writing to the trace buffer. In this way, the Trace Buffer Write Controller performs an "or" operation of the C comparison results. An "and" reduction is not supported; unlike a design specified at the RTL level, in an HLS design, the user can not be sure that any two assignments occur simultaneously in the hardware, due to optimizations that may be performed by the HLS tool. Because it may be useful to have some data in the trace buffer *after* the trigger, it is possible to continue to store data for several additional cycles, providing a sliding window of data around the point of interest selected by the user.

The motivation for basing each condition on a particular assignment rather than a variable value is due to the fact that many FPGA-based HLS tools (including LegUp) use a Static Single Assignment

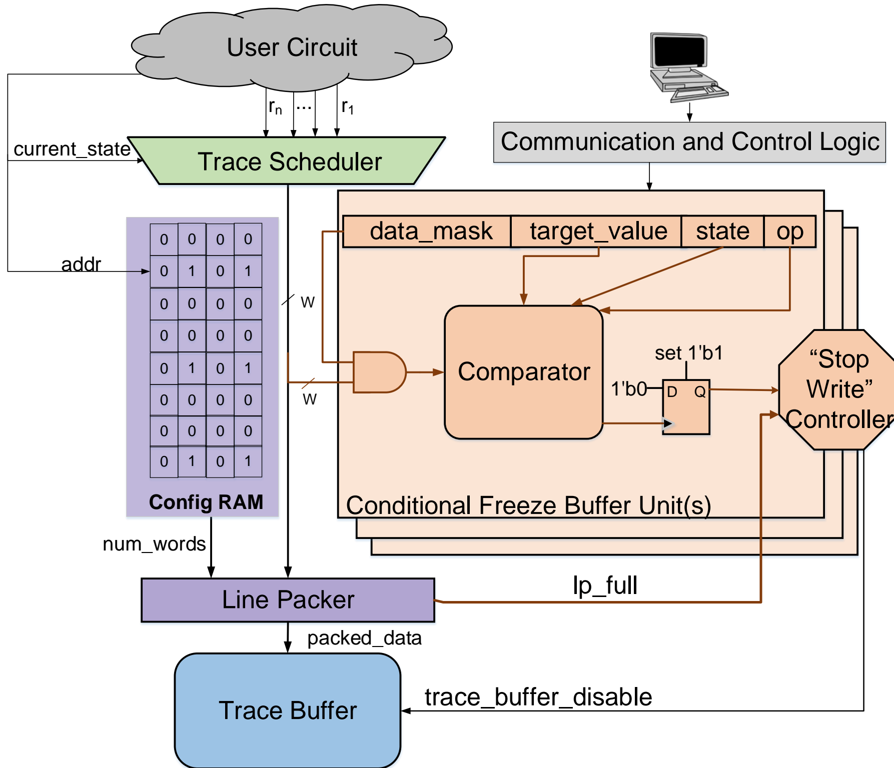


Fig. 9. Conditional Buffer Freeze Architecture

(SSA) form of the circuit's Intermediate Representation (IR). Because of the relatively high area required to implement multiplexers in an FPGA, HLS tools that target FPGAs often build hardware with distinct registers for each SSA IR assignment. If we were to monitor a variable, independent of a specific assignment, we would have to build a multiplexer to select the current value from all hardware registers corresponding to that variable. Even if there was only one register associated with each variable, since, at compile time, we do not know which variable the user would select, we would have to build a multiplexer to select from among all user variables in the circuit, which would be prohibitive in terms of area. By reusing the trace multiplexer, our overlay suffers much less overhead.

There is an extra complexity if the selected assignment is part of a function that the HLS tool inlines into multiple parents. In that case, it is impossible to uniquely identify which copy of the assignment should be used to perform the comparison. Our approach is to expose this complexity to the user if it occurs, asking him or her to identify the specific instantiation of the inlined function that is of interest. It is important to note that from a user's perspective, setting a condition can be very user-friendly in the debugger GUI interface (i.e. clicking on a line of source code and entering a condition on the variable of interest).

5 DEPENDENCY ANALYSIS

This section presents our approach for using dependency analysis to accelerate HLS debug. While the debug overlay described in the previous section can rapidly change which variables are recorded during execution, it may still be time consuming for the designer to narrow down and select which set of variables are important to observe at each debug turn. Using program analysis, we show that we can automatically determine which variables might be responsible for some observed incorrect output, and thus automatically narrow the set of variables that need to be observed.

In some ways, this is comparable to the use of dependency analysis by Yang in [24] to assist debugging. In that work, the authors provide a framework for automatic bug detection by looking for discrepancies between the software and hardware (discussed in Section 4.1). Upon locating a discrepancy, the tool performs RTL simulation of the instructions leading up to the discrepancy. By using dependency analysis to only record relevant signals leading up to the discrepancy, the authors achieve a 1.6x increase in simulation speed.

While we use similar techniques to identify dependencies in the LLVM code, our work focuses on in-system debugging rather than simulation. By leveraging dependency analysis to reduce the number of signals that need to be traced, the limited trace memory can be used more efficiently, and a longer execution trace can be captured. This, in conjunction with the debug overlay, provides guided, rapid debug turns.

5.1 Data Dependency Analysis

Data dependency analysis is a well-studied and often used technique in the software community, and is leveraged by many computer optimization techniques. The contributions in this section build on the idea of using data dependency analysis to build *program slices*, a concept first introduced by Weiser in 1981 [22]. This work recognized that when observing some value of interest in a program, there was a subset of the program that would be responsible for affecting this value of interest. This *program slice* was not necessarily the instructions immediately preceding the statement of interest, but would consist of a set of instructions, perhaps reaching back to early in the program code. These program slices are constructed using a static analysis of the program, where the goal is to identify all statements that *may* affect a value of interest. By looking at all of the available data dependencies, the programmer should be able to see what operation impacted the current value and identify what parts of the design need to be reworked.

This section describes our implementation of program slicing techniques. We describe the analysis in the context of the LLVM compiler framework, as this is the front-end used by Vivado HLS, LegUp, and most other HLS tools. This section does not provide a comprehensive description of dependencies covered by program slicing, as that is available in other works, such as [22]. Rather, we briefly provide examples of different dependency types we support, to contextualize our results.

It is important to recognize that we perform dependency analysis at the LLVM IR level, not at the C source code level. This allows us to leverage the single-static assignment (SSA) property of the LLVM IR code, and perform stricter dependency analysis than what would be possible otherwise.

```
1  x = a + calculate(b);
2  y = x++;
3  ...
4  x = c * d;
5  out = x + sum;
```

Fig. 10. Example code benefiting from dependency analysis

```
1  if (condition)
2    x = a + b;
3  else
4    x = c + d;
5  y = x;
```

```
1    %cmp = icmp ne %condition, 0
2    br %cmp label case1, label case2
3  case1:
4    %x.1 = add %a, %b
5    br end
6  case2:
7    %x.2 = add %c, %d
8    br end
9  end:
10   %y = phi [%x.1, case1], [%x.2, case2]
```

Fig. 11. Example code benefiting from SSA-level dependency analysis

For example, consider the conceptual code in Figure 10. In this case variable *x* is assigned a value at line 1, and later overwritten at line 4. When this code is compiled to LLVM IR, the SSA form will ensure that these two assignments to *x* are treated as separate variables. Thus, if the user observed an incorrect value on variable *out*, the dependencies will be identified as *sum*, *c*, and *d*, and the analysis will correctly realize that *a* and *b* are not relevant to the output.

5.1.1 Direct Dependencies. The result of computation operations in the same basic block. For example, in Figure 10, the variable *out* is assigned the sum of *x* and *sum* on line 5, and thus *out* is directly dependent on the value of these variables.

5.1.2 Control-Flow Dependencies. Figure 11 shows code where the dependencies of variable *y* are conditional upon the control flow of the program. This figure shows the C code, and the associated LLVM IR code. In such cases, LLVM uses *PHI* operations to indicate that the value of *y* may take on the value of *x.1* or *x.2*, depending on the control flow of the program.

In this case, *y* becomes *conditionally dependent* on both *x.1* and *x.2*. For this work, where we are interested in selecting which signals in an HLS circuit to record, this has the implication that the signals corresponding to both *x.1* and *x.2* will need to be recorded. Once the trace data is retrieved from the FPGA it can be analyzed to determine which conditional dependency was relevant for the captured execution. Resolving the dynamic program slice and determining which conditional dependency results in an actual dependency takes place behind the scenes, and when the user elects to navigate through the trace data they can be automatically taken to the actual dependency for the captured execution.

In theory, it may be possible to do this in hardware. If we were to design the trace scheduler to only record certain updates to the conditional variable based on the current state, we could record only relevant updates to the variable, saving on storage in our recording buffer. However, as discussed in [13], a significant portion of the debug overlay's area overhead comes from the multiplexing logic in the trace scheduler, especially in larger designs. Because of this, it is likely that this addition to the trace scheduler would result in too much area overhead. This could potentially be explored with further study.

5.1.3 Memory Dependencies. When a value is obtained from a load operation, there is a dependence on the contents of the memory, which in turn depends on store operations that would have occurred earlier in the program execution. To handle this we make use of points-to analysis to determine which arrays (and thus physical memories in the RTL circuit) each load and store instruction in the program accesses. A points-to analysis pass is included as part of LegUp 4.0, and we use this for our proof-of-concept and experiments.

Because at compile time we lack information such as the value of array indexes, we take a conservative approach and assume that all accesses to the same physical memory array may alias. As such, all loads from a memory are assumed to be conditionally dependent on any store to the memory. This is still a better result than program slicing for general software applications, as HLS tools will typically be aggressive in partitioning variables into different physical memories (for memory bandwidth benefits), meaning that it can be assured that loads from one physical memory could only be dependent on store operations that access that same physical memory.

Due to pointers, a single load or store could access multiple memories. For signal tracing implications, this means that we have to identify any memory a load could access, and then observe any store operations that could access any of these memories. As one might expect, this can cause the dependency tree to expand rapidly.

Newer versions of LLVM contain more robust memory analyses that may allow for a more aggresive approach in narrowing down which memory operations a particular memory operation is dependent on; however, our experimental testing is limited to the analysis included in LegUp 4.0, which uses LLVM 3.5.

5.1.4 Argument Dependencies. When looking at dependencies involving function arguments, we are interested both in dependencies to the function arguments (i.e. what instructions in the function are dependent on the arguments) as well as dependencies from the arguments to other instructions (i.e. at the function callsite, what variables or expressions are used as function parameters). In cases where a function is called from multiple locations, the arguments to the function will be *conditionally dependent* on the different callsites. Again, we cannot resolve the dependency at compile time, so we are forced to consider all possible callers to the function.

5.1.5 Return Dependencies. The same process occurs at the return of the function. The instructions that use the return value of the function must be noted so that a dependency relationship can be created. However, in the case where a function has multiple return statements, each of these must become a dependency on any instruction that uses the return value of the function. Again, these represent *conditional dependencies* that will be fully resolved once the execution trace is captured and analyzed.

If the user identifies a problem with a given variable, we automatically use the full set of dependencies (all types) to help guide user in selecting which variables to observe at the next debug turn. While it may be interesting to observe which types of dependencies occur most often, or which types of dependencies tend to introduce a greater overhead cost to observe, the approach we take in this work is to simply identify and use *all* dependencies of a given signal.

6 RESULTS

We first present results that quantify the overhead and capabilities of our overlay architecture. We then present results that evaluate the ability of our dependency analysis technique to identify variables that should be traced. In this work, we do not attempt to quantify the overall time to find the root cause of a bug, since that is very circuit and designer-specific. Ideally, we would conduct a user study to compare how long designers take to discover bugs with and without our tools. However, this would be quite complex to orchestrate, as we would need to design circuits with

Table 1. Trace Window Length Results

Benchmark	Baseline [13]	Full Recompile [3]		Overlay Variant A (This work)	
	100%	50%	25%	50%	25%
adpcm	2247	2876	3562	2412	2870
aes	3650	8972	17165	5762	7321
blowfish	6113	8266	10525	6873	9482
dfadd	1047	1366	1822	1056	1095
dfdiv	3391	4363	5073	3461	3490
dfmul	960	1169	1458	1043	1145
dfsin	2101	2410	2970	2164	2230
gsm	386	1597	2391	386	386
jpeg	2201	3638	4652	2233	2285
mips	739	1104	1949	739	739
motion	6212	6823	8771	6222	6229
sha	3574	6702	9431	3739	3790
FFT	636	1324	1627	662	675
Geomean	1860	2967	4035	1991	2144
vs 100%		1.60x	2.17x	1.07x	1.15x

Note: Percentages show fraction of variables selected for tracing.

sufficiently subtle bugs in them to make the debug tools worth while as well as find a significant number of engineers who could participate in a study.

Instead, we show that we can provide designers with significantly greater visibility into the design at each debug turn. This is done by increasing the amount of data that can be captured at each debug turn, combined with the expectation that the dependency analysis will help designers capture more relevant trace data, and more easily analyze the captured data. The results in this section demonstrate that both the amount of trace data captured is increased, and the dependency analysis can often reduce the number of variables that need to be observed.

6.1 Overlay Architecture Results

The architectural variants described in this paper all allow the user to trade-off the trace window size and area overhead, with essentially zero compile time between debug iterations. These abilities enable the data dependency-driven debug process described in Section 5, ideally providing the designer with faster debug turns.

In this subsection, we present the overhead cost to enable these debug features, and explore the impact of the different variants and architectural parameters.

6.1.1 Selective Variable Trace: Variant A. We first evaluate basic selective variable tracing architecture (Variant A – without the line packer) shown in Figure 5 and compare it to previous work where the trace scheduler is recompiled between each debug iteration [3, 13]. Table 1 shows the impact on trace window size as we vary the proportion of user-visible variables that are traced. To gather these results, we used the CHStone benchmark suite and the FFT_Transpose benchmark from MachSuite [15, 20]. For each benchmark, we simulated the design using Modelsim and measured the number of cycles that are stored within the trace window, averaged over the run of the program

Benchmark	Baseline	Variant A		Variant B (G=2)		Variant B (G=4)		Variant B (G=8)		Variant B (G=16)	
		ALMs	Diff	ALMs	Diff	ALMs	Diff	ALMs	Diff	ALMs	Diff
adpcm	11717	11684.2	-33	12106.2	389	12320.2	603	13264.2	1547	15477.8	3761
aes	10083	10110.4	27	10432	349	10428.2	345	11551	1468	13414.4	3331
blowfish	5426	5407.2	-19	5461.8	36	5606.6	180	6058.4	632	7421	1995
dfadd	6808	6864.4	56	7773.4	965	7832.2	1024	16315.6	9507	38191.4	31383
dfdiv	9277	9403	126	9941.4	664	10538.6	1261	21706.2	12429	50686.6	41409
dfmul	4205	4181.8	-23	4804.6	599	5211.2	1006	13846.6	9641	40532	36327
dfsin	19165	19546.8	381	20155.4	990	21050.6	1885	22668.2	3503	62200.2	43035
gsm	7333	7441	108	8036.6	704	9351.2	2018	19363.2	12030	15640.8	8308
jpeg	26985	26837.4	-148	27677	692	28405.6	1420	29745.2	2760	69522.2	42537
mips	3740	3811	71	4678	938	5531.2	1791	6830.2	3090	10767.2	7027
motion	9530	9639	109	9775	245	9946.2	416	10444	914	11828.6	2299
sha	3017	3017	0	3317	300	3531.6	515	6111	3094	16216.4	13200
FFT	69036	69135	99	71123.6	2087	71868.2	2832	73415.4	4379	77466.4	8430

Table 2

(a higher number means that the trace buffer is being used more efficiently, and that there is more information available to the off-line debugging GUI). In all cases, a 100Kb trace buffer was assumed. In selecting a subset of variables to observe, we selected variables randomly; we average the results over five runs with different seeds to minimize the impact of especially bad or good variable selections.

In this table, Columns 2-4 show the results for the baseline architecture in which the trace scheduler is compiled between each debug iteration. Columns 5-6 show the results for our architecture. From this table, we can observe that while recording fewer variables does increase trace length, the benefit is much larger when a full recompile is performed (1.60−2.17x), versus using the first variant of our overlay (1.07−1.15x). While our overlay provides much faster turnaround times, this variant has limited benefit to the trace length (this is addressed by the line packer in Variant B).

The other price we pay for software-like turn-around times is area. To measure area, we instrumented each benchmark circuit and mapped the results to a Stratix IV FPGA. We report post place-and-route numbers to account for any physical design optimizations that Quartus II is able to perform. Over all benchmarks, the average area of the baseline architecture (based on the previous work [13]) was 2232 ALMs, not including the trace buffer itself. Our enhanced architecture, Variant A, required a total of 2271 ALMs plus one M9K memory block (this is an increase of 39 ALMs and one memory block). Of the 39 ALMs increase, 59% (on average) was due to logic required to make the memory block accessible using Intel's In-System Memory Content Editor [1].

We found little impact on the maximum clock speed of the circuit, ranging from -9% to +12% (average +1%), which we attribute to algorithmic noise in the CAD tool.

Even though our technique suffers in terms of area and trace window length, for our benchmarks, the time to personalize the overlay ranges from 0.887 seconds to 1.751 seconds (dominated by the time to update the ROM via the memory content editor). In contrast, the approach in [13] suffers a compile time of 454 seconds on average for the same benchmarks. In [3], this is improved to 231 seconds (assuming 50% traced signals) using Intel's incremental compilation flow in Quartus II v16.0.

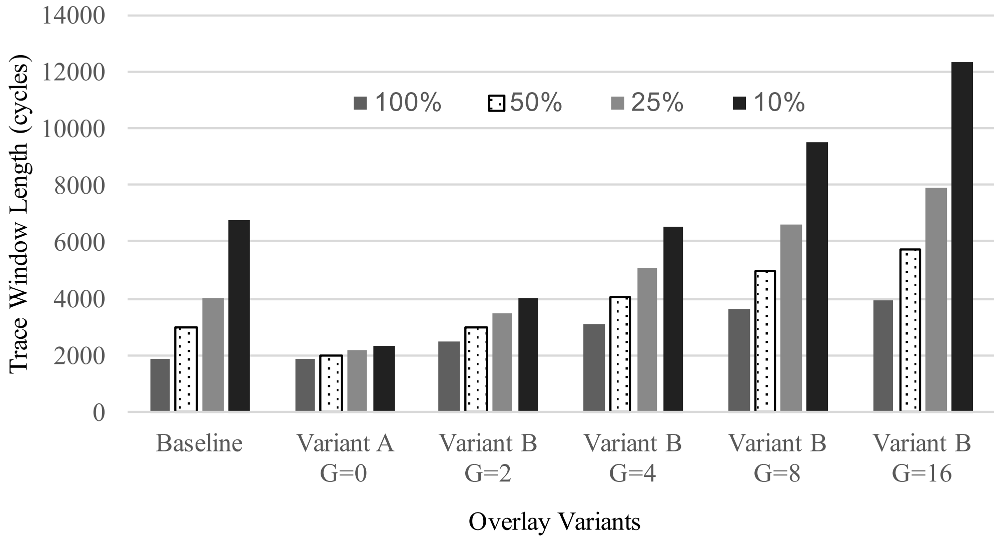


Fig. 12. Impact of Line Packer Granularity (G) on Trace Window Size - Variant B

6.1.2 Impact of Line Packer: Variant B. To improve the trace buffer capacity, Section 4.3 proposes Variant B, which contains the line packer module. Figure 12 shows the impact of the trace length as a function of the line packer's granularity (G) for this variant. Compared to the trace lengths achieved by the baseline and Variant A architectures, G=2 reclaims the trace length lost with Variant A and G=16 performs 2x-7x better than the baseline. Figure 13 shows the impact of the overhead logic area versus the baseline as a function of G (all implementations also require one M9K memory block). This graph depicts the additional overhead required for our line packer variants compared to the baseline described in [13]. This overhead is dominated by the line packer (Figure 8) and is thus primarily a fixed overhead, independent of the user's design. As the graphs show, increasing G has a significant impact on the trace length. The area, on the other hand, grows more quickly as G increases, primarily due to the increased area of the steering logic.

Compared to Variant A, this variant requires more memory bits, however, for all benchmarks, only one M9K block was required to store all configuration bits, even for G=16.

Once again, we saw only a small impact on the clock frequency, except for the Variant B architecture where G=16 as shown in Figure 13. This is due to the increased chain of steering logic for a higher granularity line packer. At lower G, the frequency dropped slightly as compared to the baseline. The critical path of our instrumented circuits typically fall in the user circuit itself rather than the instrumentation. If we were instrumenting an extremely high-frequency circuit, we could pipeline our instrumentation to match the clock speed.

6.1.3 Selective Function Tracing. As described in Section 4.4, adding the selective function tracing capability may allow a user to use trace buffer space more efficiently by focusing on specific functions of interest. The benefit of this capability on trace buffer length is clearly very circuit-dependent and function-dependent, and thus overall averages may not be meaningful. However, as a data-point, we gathered the results in Table 3. This table shows the impact of tracing two functions on trace window size for a single benchmark circuit, *adpcm*, for two variants of the

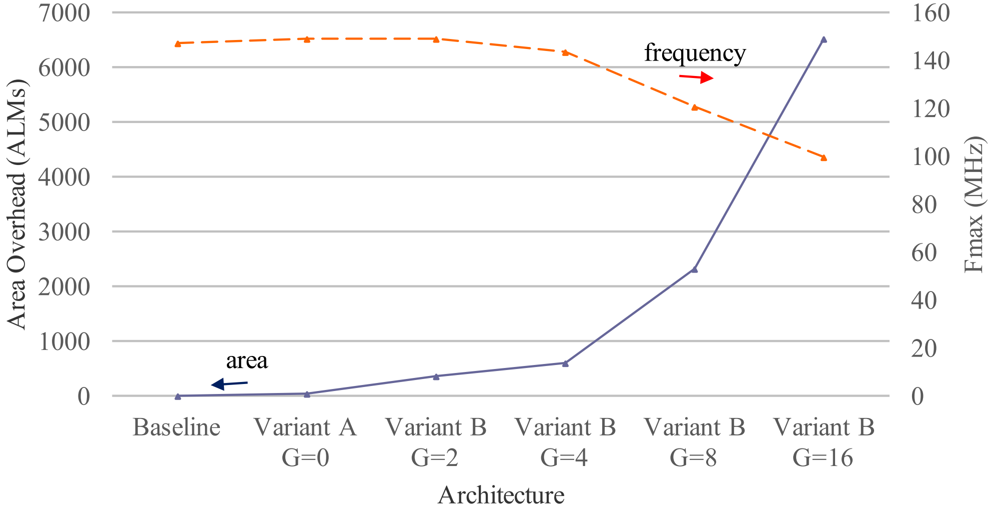


Fig. 13. Impact of Line Packer Granularity (G) on Area - Variant B

overlay mapping algorithm: *partial control flow* where control flow information that is outside the selected function is not recorded, and *full control flow* where all control flow information is recorded, regardless of whether it is outside the traced function. While these two variants change whether control flow information is traced outside the selected function, in both cases we are only tracing data information from the function of interest. In this experiment, the baseline trace window size (where everything is traced) was 2613 cycles. When we only trace the *encode* function, this rises to 4024 cycles for the full control flow method, and 4278 for the partial control flow method. For the *upzero* function, the trace window size is 4932 cycles for the full control flow method and 6532 cycles for the partial control flow method. Clearly, *upzero* benefits more from using partial control flow. One reason is that *encode* is inlined by the HLS tool; in this situation, we are conservative and trace the entire parent function. Thus, we would expect that the benefits of not tracing outside the selected function (in this case, the parent function) would be smaller than a function that is not inlined, such as *upzero*.

As described in Section 4, there is no area impact in supporting the selective function tracing capability.

6.1.4 Conditional Buffer Freeze. The area impact of the conditional buffer freeze capability is shown in Figure 14 as a function of C (the number of subunits, which affects the complexity of the conditional function that can be used). The left vertical axis is the increase in area when this capability is added to Variant B. As the graph shows, as C increases, the overall area increases significantly. As before, we saw negligible impact on clock speed as C increases.

Table 3. Control Flow Tracing Results for adpcm

Function traced	Inlined?	Full Control Flow	Partial Control Flow
none		2613	2613
encode only	yes	4024	4278
upzero only	no	4932	6532

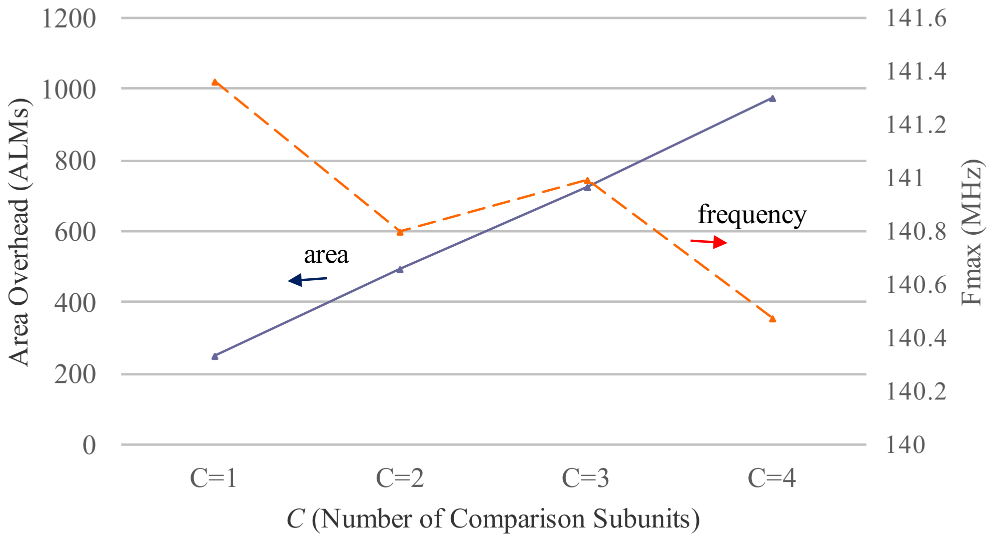


Fig. 14. Impact of Number of Sub-units (C) on Area for Conditional Buffer Freeze Architecture

The conditional buffer freeze capability has no impact on achievable trace length, but rather the capability is meant to enable the designer to capture critical information in the trace buffer that otherwise could not as easily be captured. This is difficult to quantify as it would be very specific to a particular circuit, designer, and bug. However, the primary motivation of this feature is that without adding some conditions to a buffer freeze point, it would by necessity be triggered the first time execution reaches the user-chosen line of code. In HLS, where much of the code is spent within loops, it is feasible that the small hardware trace buffers would be filled multiple times by one loop instance. This is problematic, as the designer would be limited to placing a freeze point at the beginning of a loop and capturing early iterations, or after a loop terminates and capturing later iterations. If a designer wanted to observe iterations toward the middle of execution, there would be no way to capture the desired data. By providing the conditional freeze capability, a designer can more effectively narrow the captured data on specific loop iterations. This is just one of many examples of its use. Another case might be if a function were called multiple times, and a designer were interested in tracing only invocations of a function under a certain condition. Again, targeting such specific trace data would be very difficult without conditional freeze points.

6.1.5 Overlay Results Summary and Discussion. Overall, these results show the trade-off between trace window size and area overhead for the various capabilities we have discussed, along with the impact of various architectural parameters. These results are likely to be useful to an FPGA vendor who wishes to create an ecosystem containing an overlay such as ours. In such an ecosystem, the HLS tool could determine, based on an estimate of the amount of space left on the FPGA, how large an overlay to construct. If it is estimated that there is little space available, it might construct an "economy" version with only one or two capabilities. If more space is available, it may choose to construct a "deluxe" version that supports all capabilities we have discussed (and perhaps others). The results showed that the cost of our Selective Trace architecture is very small (Variant A required only 39 ALMs and one M9K block beyond the baseline), and that, when the user can be selective in which variables should be traced, the impact of including this capability on

trace length is significant. This suggests that almost any overlay should likely have at least this capability. The conditional buffer freeze capability is more expensive (between 200 and 1000 ALMs depending on the complexity of the condition supported), so an automatic tool might only choose to include this capability in the overlay if there is sufficient space available.

We have purposefully not addressed the question of how useful each capability is in finding a bug (other than the extent to which some capabilities increase the trace window size). Understanding the trade-off between increased window size and improved buffer freeze capabilities, for example, would require user studies or interactions with a large user base that could provide experience and insight into what features help them find their most complex bugs.

6.2 Effectiveness of Dependency Analysis

As explained in Section 3, the primary uses cases we see for dependency analysis in HLS debug are 1) to ease the user experience in performing root-cause analysis of the trace data, and 2) to automate the variable selection process for the next debug turn. An evaluation of the benefits to user experience is challenging; it is not clear how one would quantify the ease by which a designer can analyze the trace information. Rather, we focus on the latter, and quantify the benefit of narrowing the signals that need to be observed.

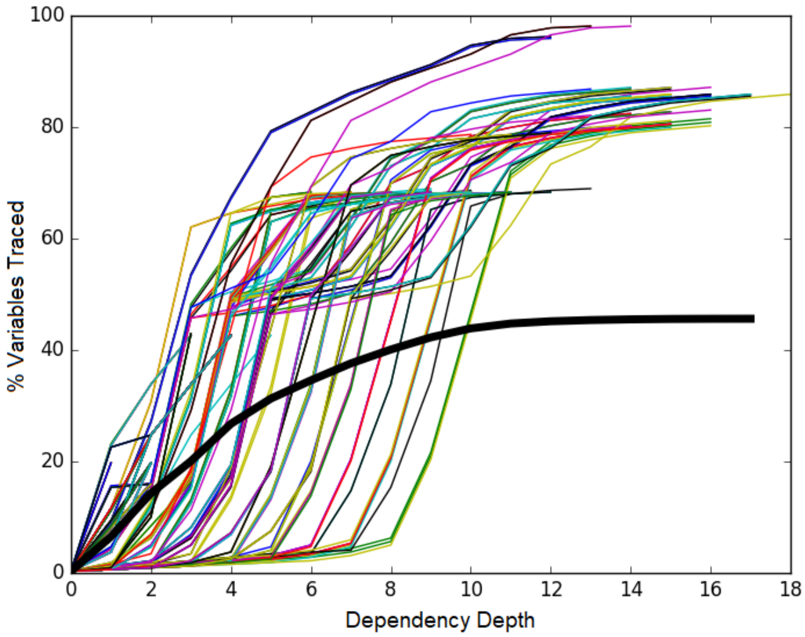
When the user identifies a problematic variable, one could elect to observe *all* upstream dependencies, however, for certain variables in the program, especially outputs, this will likely result in observing all, or nearly all of the variables. Instead, we introduce the concept of ***Dependency Depth***, which provides a measure of how many dependency hops back in the tree one elects to observe. In normal debugging scenarios, we expect a user would only elect to trace a handful of steps back in the dependency tree.

For example, in Figure 10, if the *out* variable is identified as problematic, a dependency depth of zero would represent only observing *out*, while a depth of one would include *x* and *sum*, and a depth of two would include *c* and *d*. It is important to recognize that due to conditional dependencies from control flow and memory accesses, it is possible for just a few hops back in the dependency tree to quickly grow to include large portions of the program variables in the set of relevant variables.

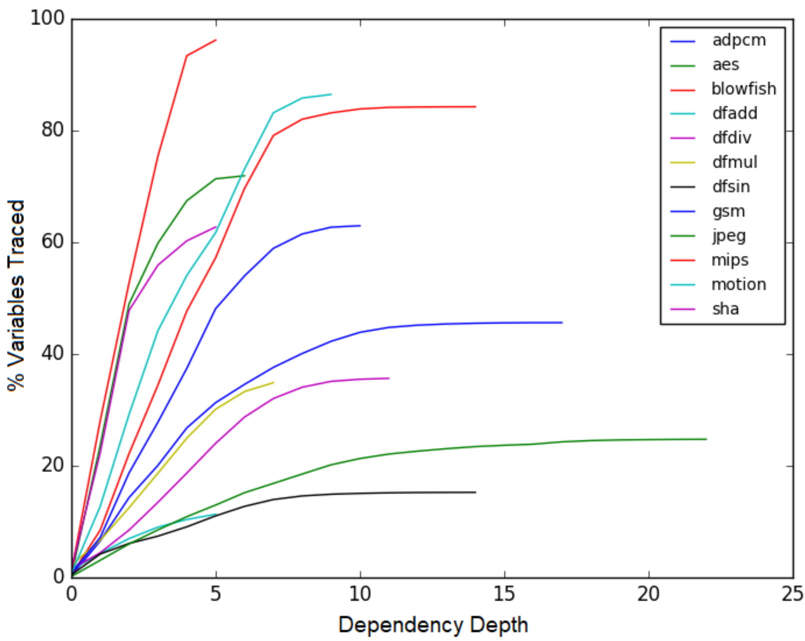
Assuming the user selected some variable in a program as problematic, we wanted to measure what portion of the program variables would be identified as potential dependencies, and would necessitate recording. If the set of dependencies is relatively small, then many variables could be ignored, and the benefit to the user would be significant. On the other hand, if just a few levels of dependency depth resulted in nearly all of the variables in the program being marked as dependencies, then our technique would have limited value.

Since the results are highly dependent on which variable the user marks as problematic, we elected to explore all possible cases for each of our benchmarks. That is, for each of the benchmarks we tested the effect of dependency driven signal selection, originating at each of the variables in the program. Our experiments were performed using a modified version of LegUp 4.0 [6], and using the CHStone HLS benchmark suite [15]. Across the 12 benchmarks, we identified 2033 variable assignments in the C code that could be chosen as the starting point for dependency analysis. From these starting points, we explored a dependency depth of zero, all the way up to the point where the depth saturates and all of the variable's dependencies were marked for recording. Overall this resulted in 15011 data points.

Figure 15 shows the impact of using dependency analysis to select which variables will be observed. Figure 15a shows the data for only the *adpcm* benchmark. Each line represents one of the 319 potential starting points for dependency analysis that the user could select. For each starting point the dependency depth is swept from 0 (recording only the identified variable) to the depth that would result in all of its dependencies being observed. The thick black line represents the

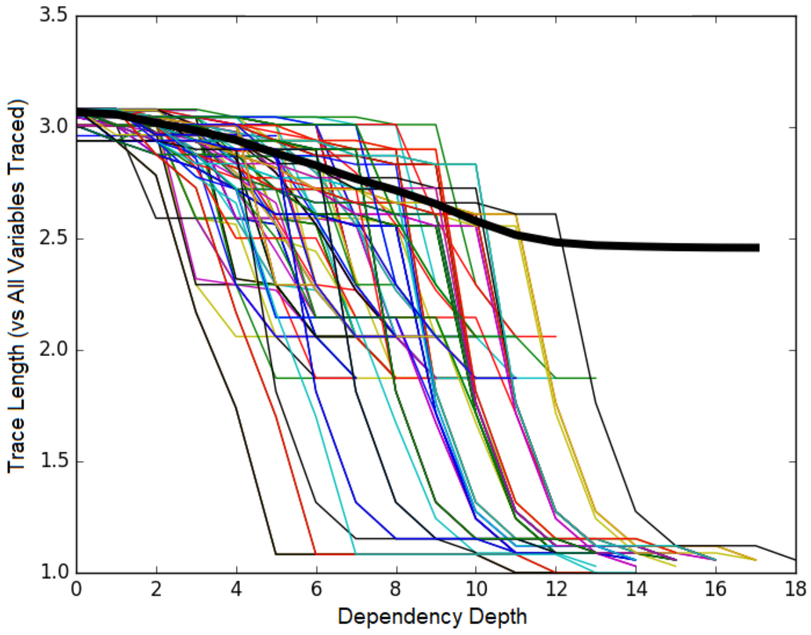


(a) ADPCM Benchmark

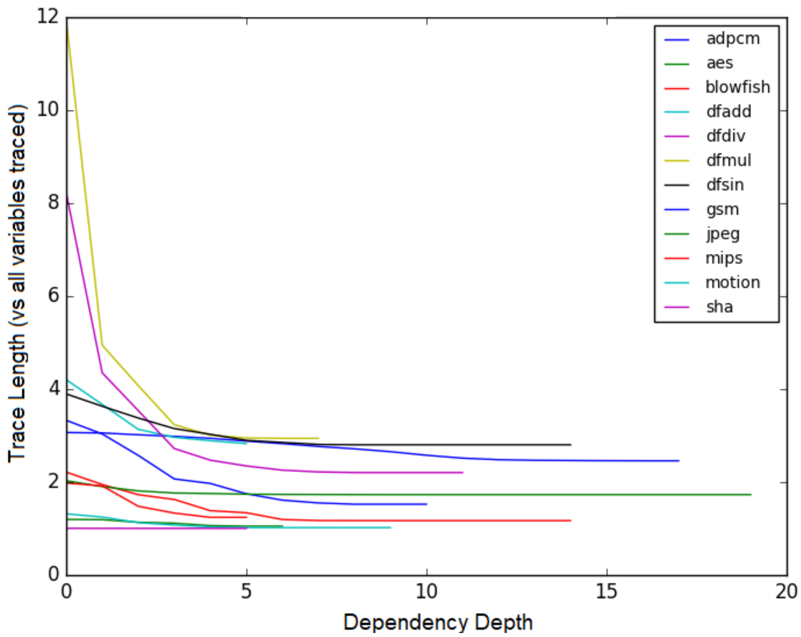


(b) All CHStone benchmarks

Fig. 15. Effect of dependency-based variable selection on the number of variables that need to be observed



(a) ADPCM Benchmark



(b) All CHStone benchmarks

Fig. 16. Effect of dependency-based variable selection on the average captured trace length

average of the 319 cases. As shown in the results, only a few of the variables are dependent on all, or nearly all variables in the program; these would correspond to the final outputs of the program. While these may be the starting point for the first debug turn, we anticipate users will quickly narrow the problem to one part of the program, and will observe variable dependency trees that make up only a portion of the total variables. The results show that on average, for the *adpcm* benchmark, variables are dependent on less than half of the program variables, and this drops even lower if the dependency depth is limited.

Due to limited space, it is not feasible to include these figures for each of the 12 CHStone benchmarks; instead, Figure 15b shows the average curve for each of the benchmarks. As can be seen, *adpcm* is a typical case; for some designs the dependencies quickly encompass nearly all program variables, and these analysis techniques will have limited benefit. For others, individual variables, on average, are dependent on a minority of the variables in a program, and will benefit more greatly.

6.3 Trace Length

Ultimately, reducing the number of variables that need to be observed is only a benefit if this will result in the designer being able to capture longer execution traces. To measure this, we simulated the execution of each of our 15011 scenarios in Modelsim, and measured how many cycles it took on average to fill the trace buffers. This gives us a representative value of the execution length that a user would be able to capture. This execution trace duration was then compared against a baseline case where the user elects to simply record *all* variable updates. The ratio results are provided in Figure 16. Again, the top figure illustrates the result for only the *adpcm* benchmark, and the bottom figure shows the average trends for all of the CHStone benchmarks. For *adpcm*, the average trace length is 2.5–3x longer than if the user simply elected to capture all variable updates. This increased execution length that is captured will hopefully make it easier for the designer to locate the root cause of a bug, and will reduce the number of debug turns.

Across all of the CHStone benchmarks, we see that the benefit varies. Some designs exhibit large (5–10x) increases in trace length for smaller dependency depths, while others see little benefit from this approach.

7 CONCLUSIONS

In this work we demonstrated techniques to provide faster turnaround time for in-system HLS debugging, where dependency analysis was leveraged to guide users in selecting variables to observe at each debug iteration, as well as a debug overlay that provided rapid reconfiguration of the debug circuitry to allow the user to change which variables are observed without needing a long-running FPGA recompile. The architecture study showed a meaningful tradeoff in debugging capabilities that can be provided to the designer and overhead cost to enable these features. We believe that the techniques and results presented here would be valuable information for FPGA vendors that wish to create an accelerated in-system HLS debug system, such as ours.

REFERENCES

[1] Altera. 2016. Altera Virtual JTAG (altera_virtual_jtag) IP Core User Guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_virtualjtag.pdf.

[2] Altera. 2016. SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[3] P.K. Bussa, J. Goeders, and S.J.E. Wilton. 2017. Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation techniques. In *International Conference on Field-Programmable Logic and Applications.*

[4] N. Calagar, S.D. Brown, and J.H. Anderson. 2014. Source-level Debugging for FPGA High-Level Synthesis. In *International Conference on Field Programmable Logic and Applications.*

[5] Keith Campbell, Leon He, Liwei Yang, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. Debugging and Verifying SoC Designs Through Effective Cross-layer Hardware-software Co-simulation. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2897937.2898002

[6] A. Canis, J. Choi, et al. 2013. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems* 13, 2, Article 24 (Sept. 2013), 27 pages.

[7] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. 2009. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers* 26, 4 (July 2009), 8–17. https://doi.org/10.1109/MDT.2009.69

[8] F. Eslami and S. J. E. Wilton. 2015. An adaptive virtual overlay for fast trigger insertion for FPGA debug. In *International Conference on Field Programmable Technology*. 32–39.

[9] P. Fezzardi, M. Castellana, and F. Ferrandi. 2015. Trace-based automated logical debugging for high-level synthesis generated circuits. In *International Conference on Computer Design*. 251–258. https://doi.org/10.1109/ICCD.2015.7357111

[10] P. Fezzardi and F. Ferrandi. 2016. Automated bug detection for pointers and memory accesses in High-Level Synthesis compilers. In *International Conference on Field Programmable Logic and Applications*. 1–9. https://doi.org/10.1109/FPL.2016.7577369

[11] Pietro Fezzardi, Marco Lattuada, and Fabrizio Ferrandi. 2017. Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 149 (Sept. 2017), 19 pages. https://doi.org/10.1145/3126564

[12] J. Goeders and S.J.E. Wilton. 2014. Effective FPGA debug for high-level synthesis generated circuits. In *International Conference on Field Programmable Logic and Applications*. https://doi.org/10.1109/FPL.2014.6927498

[13] J. Goeders and S.J.E. Wilton. 2017. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 1 (Jan 2017), 83–96.

[14] J. Goeders and S.J. E. Wilton. 2015. Allowing Software Developers to Debug HLS Hardware. In *Workshop on FPGAs for Software Programmers*.

[15] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. 2009. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing* 17 (2009), 242–254.

[16] AS. Jamal, J. Goeders, and S.J.E. Wilton. 2018. Architecture Exploration for HLS-Oriented FPGA Debug Overlays. In *International Symposium on Field-Programmable Gate Arrays*.

[17] J. S. Monson and B. Hutchings. 2014. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In *International Conference on Field-Programmable Logic and Applications*. 1–6.

[18] J. S. Monson and Brad L. Hutchings. 2015. Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs. In *International Symposium on Field-Programmable Gate Arrays*. 5–8.

[19] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* PP, 99 (2016). https://doi.org/10.1109/TCAD.2015.2513673

[20] B. Reagen, R. Adolf, Y.S. Shao, Gu-Yeon Wai, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *International Symposium on Workload Characterization*. 110–119.

[21] Hayden Kwok-Hay So and Cheng Liu. 2016. *FPGA Overlays*. Springer International Publishing, Cham, 285–305. https://doi.org/10.1007/978-3-319-26408-0_16

[22] Mark Weiser. 1981. Program slicing. In *International conference on Software engineering*. 439–449.

[23] Xilinx. 2016. Vivado Design Suite User Guide: High-Level Synthesis. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf.

[24] Liwei Yang, Swathi Gurumani, Deming Chen, and Kyle Rupnow. 2016. AutoSLIDE: Automatic Source-Level Instrumentation and Debugging. In *International Symposium on Field-Programmable Custom Computing Machines*. 127–130. https://doi.org/10.1109/FCCM.2016.38

[25] Liwei Yang, Swathi Gurumani, Suhaib A Fahmy, Deming Chen, and Kyle Rupnow. 2016. Automated Verification Code Generation in HLS Using Software Execution Traces. In *International Symposium on Field-Programmable Gate Arrays*. 278–278. https://doi.org/10.1145/2847263.2847313