Control Flow Tracing for High-Level Synthesis

Do Yoon Kim*
University of Michigan

Insu Jang*
University of Michigan

Jae-Won Chung*
University of Michigan

Jiachang Ma
University of Michigan

Abstract

High-level synthesis (HLS) is increasingly adopted as a method for programming FPGAs with high-level C/C++/OpenCL code. However, the lack of observability in hardware precludes profile-guided optimization based on production data. Thus, as a convenient method for introspecting the flow of execution inside synthesized hardware, we implement a framework that enables control flow tracing for HLS. Our LLVM pass instruments the high-level source code transparently with function calls to the tracer, which records the current location of control flow with respect to the original source code. Trace data is exported via an AXI master interface, which we also transparently add to the user module, to accessible storage such as off-chip DRAM. Our project is the first open implementation of control flow tracing for HLS, laying the groundwork for future research in HLS debugging and profile-guided optimization.

1 Introduction

High-Level Synthesis (HLS) has recently risen as a method that allows FPGAs to be programmed with high-level languages such as C, C++, or OpenCL [4, 1, 11, 12, 8]. While there existed need and motivation to accelerate software applications with FPGAs, the lack of expertise in hardware description languages (HDL) such as Verilog or VHDL often acted as a major obstacle for software developers [3]. HLS gracefully addresses this mismatch by compiling code written in high-level languages into HDL, drastically lowering the bar of utilizing FPGAs for accelerating applications.

However, FPGAs are still difficult to program due to their lack of observability [10, 13, 7]. That is, it is very difficult to introspect what's actually happening inside a programmed FPGA, and HLS in fact exacerbates this problem. In the HLS design workflow, high-level constructs written by the developer are synthesized into primitive HDL code not meant to be read by humans, thereby adding a layer of obscurity to the traditional hardware design workflow. Moreover, many HLS frameworks make performance optimizations such as loop pipelining and array partitioning, which further dissolve the shape of the original high-level source code.

Nevertheless, being able to observe execution directly in hardware is much needed for two reasons. First, profile-guided optimizations (PGO) [6, 14, 2], where the implementation of an application is optimized based on profile information, are only effective when performed based on production data. However, production data only flow into hardware that are already deployed and running, and storing data elsewhere for the purpose of future analysis can quickly become too expensive to be practical. Thus, a mechanism that allows the collection of profile information (e.g., branch probability, loop trip count) from running hardware has significant benefit. Second, there are discrepancies between high-level implementation and synthesized hardware that only surface after HLS (e.g., the initial value of accidentally uninitialized variables, unexpectedly created local array initialization loops, control flow before and after static optimizations such as dead code elimination). Thus, observing what is actually synthesized and ran on hardware provides the developer with a more accurate view of what's happening inside.

In this work, we aim to answer such need by designing and implementing an HLS extension that can collect control flow trace directly on hardware. Given the high-level source code written by the developer, our framework transparently instruments it with calls to the tracer, which collects the current location of control flow with respect to the high-level source code. Our framework solves the two aforementioned problems in that the control flow trace collected on hardware can be analyzed to generate profile information, and since the trace collected is represented in terms of the debug information

*equal contribution, name in alphabetical order

of the high-level source code, providing developers with a precise view of the actual control flow inside the programmed hardware.

The correctness of our framework is verified by cosimulating [8] the synthesized Verilog design alongside with a C test bench that parses raw trace data into JSON. Moreover, the latency and resource (area) overhead of our framework is measured and analyzed. Finally, we demonstrate the usefulness of our framework by providing a compelling case study: Automatic exploration of loop unrolling resource efficiency. Specifically, we identify the hottest loop from user modules from the control flow trace collected with our framework. Then, we automatically explore multiple loop unroll factors and plot the performance gain per resource usage. Examining this plot, developers can make more informed choices about loop unroll factors, a choice once made through manual exploration or guessing.

The rest of this paper is organized as follows. We go through necessary background in section 2. Then, we present the design and implementation of our tracer framework in section 3, and show evaluation results in section 4. We provide some ideas on future extension of our framework in section 5. Finally, we conclude in section 6.

2 Background

2.1 HLS Frameworks

For better usability, the control flow tracer should be integrated with existing open source HLS frameworks. There were several requirements to be met when choosing the appropriate framework. Since tracing is implemented as an IR pass, we narrowed down the search to well-maintained frameworks where at least the front end of the compiler was available. An LLVM [9] based tool was preferred for greater applicability of our solution. Since we must be able to store the collected trace in the DRAM, it was essential that the tool provides a memory interface. Finally, to make the most out of existing tools, a framework that exposes the IR to its users and provides easy testing and simulation was preferred, though not a critical factor. We have investigated three candidates: XLS [5], Bambu [12], and Vitis HLS [8].

XLS [5] is a framework that is under active development and internally used by Google. It is completely open source, with both the front and back end available. However, XLS uses its own intermediate representation, possibly impacting the applicability of our tracer. A more critical limitation was that XLS does not provide a memory interface, making it difficult if not impossible for the control flow trace to be collected and stored.

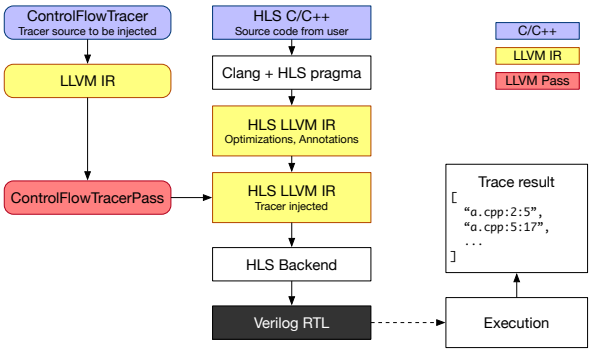

Figure 1: Architecture overview. `ControlFlowTracer` pass transparently injects tracer function calls to proper lines of user source code to trace control flow during compiling user source code.

Bambu [12] is a well-maintained LLVM based framework, which provides support for memory and convenient testing. However, since the LLVM IR is kept as an internal data structure in Bambu, applying IR passes to it would require additional efforts to extract the IR first.

The last candidate was Vitis HLS [8], an open source project provided by Xilinx. Though only the front-end of the compiler is available, Vitis HLS provides great flexibility in that it exposes the LLVM IR to its users and supports custom LLVM passes. Pre-synthesis C/C++ simulation and post-synthesis co-simulation of synthesized RTL are also supported. As Vitis HLS was the only framework that satisfied all our requirements, we decided to integrate the control flow tracer with Vitis HLS.

2.2 Control Flow Tracing

Control flow tracing refers to recording the order in which a given software is executed. The requirement of a control flow trace is that by only looking at the trace, one should be able to entirely recover the control flow of that specific program execution. In a high-level language, this corresponds to recording execution behaviors such as whether an if branch or an else branch was taken, or how loops were iterated and exited. From an LLVM perspective, these examples would be represented as decisions made from `br`, `call`, and `ret` instructions.

3 Design & Implementation

There could be several design candidates for control flow tracing in FPGA. We could implement a FPGA module that monitors which logic gates are used during execution; with FPGA's partial reconfiguration feature, we could reconfigure user logic part to reuse our module for all user programs. However, hardware-level transparent

Name	Usage
`init`	Initialize tracer metadata.
`record`	Write line number and column number of specific source code location to the RAM.
`finish`	Used before the top-level function finishes to automatically flush the recorded trace.

Table 1: List of `ControlFlowTracer` functions.

tracing is hard to implement and does not reflect properties of high-level source code. Instead, we chose to instrument user code by adding tracer function calls before implementing the corresponding HDL. This hybrid design

Our instrumentation is done in IR-level, not source-code level, for the following two reasons. First, IR is independent to the language that source code is implemented. HLS supports C, C++, and OpenCL as of now, and more languages could be supported; it is not practical to implement tracer modules for each language binding. Second, it is easier to incorporate profile information and profile guided optimization. Our architecture could be used with such information for further optimization in IR-level.

3.1 Architecture Overview

Figure 1 illustrates our architecture. When user compiles its source code to implement the corresponding HDL, the compiler feeds it to `ControlFlowTracerPass`, an LLVM pass that instruments the given source code IR and injects `ControlFlowTracer` function calls, which stores control flow information into memory. The pass extracts line number and column number from the user source code IR and transfers them to `ControlFlowTracer` functions.

3.2 `ControlFlowTracer` Functions

`ControlFlowTracerPass` LLVM pass injects functions in `ControlFlowTracer` into the user source code for control flow tracing. Table 1 explains functions that our `ControlFlowTracer` includes.

- `init()`: The tracer stores information of taken control flow as a stream of pairs of line number and column number. To store those data sequentially, we need to know the current index of RAM. `init()` initializes such tracer-internal metadata.
- `record()`: Call of this function is inserted to every point of user source code that is required to trace control flow. It stores the corresponding source code information (e.g. line number and column number) into the given RAM. It uses internal metadata what are initialized in `init()` to store data in proper location.
- `finish()`: Once program terminates, we lose RAM information where our trace data is stored. To prevent trace data loss, call of `finish()` is inserted before user's top-level function terminates to dump the recorded data.

3.3 Source Code Instrumentation

`ControlFlowTracerPass` pass transparently injects calls of `ControlFlowTracer` functions to proper location of user source code in IR-level. The injection happens when it instruments the given source code; it should determine properly to which point function calls should be added. We introduce how we find out proper locations that functions should be added for control flow tracing.

Our algorithm focuses on finding *conditional branch*, which is implemented as a `br` instruction preceded by `cmp` instruction. All control flow statements that diverge control flow (e.g. `if-else, while, for`) are internally based on the conditional branch.

```
; <label>:1:
  ...
  // if %1 < 0? then goto %2; else goto %3
  %cmp = icmp slt i32 %1, 0
  br i1 %cmp, label %2, label %3

; <label>:2: // branch taken target

; <label>:3: // branch fall through target
```

The example IR above illustrates an example of conditional branch; control flow diverges with conditional branch in the block with label %1. We focus on the following two properties of conditional branch for finding them: conditional branch IR is always generated as a consecutive sequence of `cmp` and `br` without instructions between two, and IR blocks with a conditional branch always terminate with `br`. We add `record` tracer function call at the beginning of the two successor blocks, %2 and %3 in the example, based on the two properties.

This simple algorithm, however, generates unnecessarily duplicated trace record function call injection. Figure 2 illustrates a loop and the corresponding control flow diagram. Our goal in this case is to trace whether flow reaches to either A, B, or C, but not in the red block, since if the flow reaches either B or C we would implicitly know the red block is reached without traced data.

We should avoid unnecessary record call injection to minimize performance overhead due to tracing. To remove such unnecessary injection, we first implement a

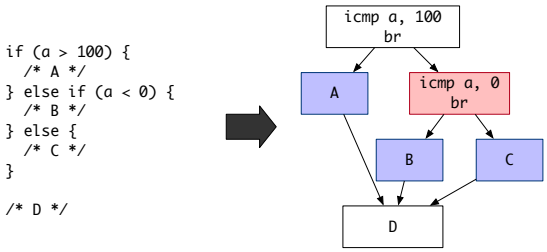

Figure 2: An example of loop with its control flow diagram.

list of candidate blocks with the successors of conditional branches, and then remove blocks if they also end with a conditional branch. This improvement actually removes unnecessary injections in not only the loop aforementioned, but also in complex combined for and if statements.

3.4 Algorithm Implementation

Algorithm 1 Tracer function call injection algorithm.

```
1:  function RUNONUSERFUNC(func)
2:      Inject init() at the beginning
3:      Inject finish() at the end
4:      cand_blks := ∅
5:      for each block b in func do
6:          if b ends with cmp & br then
7:              cand_blks ← b.successors
8:          end if
9:      end for
10:     for each block b in cand_blks do
11:         if b ends with cmp & br then
12:             cand_blks.remove(b)
13:         end if
14:     end for
15:     for each block b in cand_blks do
16:         i ← first instruction in b with source info
17:         inject record(i.line, i.col) before i
18:     end for
19: end function
```

Based on what explained in Section 3.3, we implement Algorithm 1. It first builds a list of candidate blocks, which are successors of the blocks that terminates with `cmp & br` (line 7). We use `llvm::BasicBlock::getTerminator()` to get the terminator instruction and check whether it is a branch instruction. A terminator instruction of the block is always at the end of the blocks, and `br` is one of the types of terminator. LLVM provides `llvm::BranchInst::isConditional()` function to check whether the given `br` is a conditional branch. Then to reduce duplication, we remove candidate blocks who terminate with `cmp & br` as well (line 12). For the remaining blocks, we find the first instruction with the source code information, and extract the information using `llvm::Instruction::getDebugLoc()` API. Actual function injection is implemented with `llvm::IRBuilder` class methods.

However, some IR instruction may not contain source code information, or even worse, all IR instructions in the block may not contain at all. This is because some IR blocks exists just for building optimized IR graphs. In this case, we could not extract source code location information by using `llvm::Instruction::getDebugLoc()`. To solve this problem, we borrow source code location information from its successor block, traversing blocks downward until found. We verified that this maneuver does not break any correctness of our trace information.

3.5 FPGA DRAM Interface

Our tracer could generate huge amount of trace data very fast, we should consider both capacity and performance as a media. Block RAM (BRAM) is fast, but typically to small, while I/O would be too slow. We assume a FPGA to be used has DRAM and a memory controller and store data in DRAM. HLS framework implements an AXI interface to which DRAM controller is connected, and translates all array accesses to AXI protocol signals [15].

```
int top(int dram[256]) {
  #pragma HLS INTERFACE m_axi port=dram
  ...
}
```

In HLS, top-level function could access DRAM via a pointer with pragma annotation; when annotated in source code, HLS framework would automatically generates proper interface for DRAM access that is suitable for the given information. This pointer could also be used in other functions, if top-level function passes it to the callee as an argument.

We wanted to implement our framework as transparent as possible, hence we tried to inject an array argument into the argument list of top-level function with pragma annotation. We found that such pragma had been transformed to IR attributes into IR:

```
; Function Attrs: nounwind
define i32 @_Z3topPi
(i32* "fpga.decayed.dim.hint"="256" %dram)
```

which can be extracted as:

```
auto param_attr = func.getAttributes().
getParamAttr(0, "fpga.decayed.dim.hint");
```

However, it meant that HLS framework already handles pragma by translating it into IR attributes, so we were not sure whether injecting such attributes in our IR pass would be effective. Plus, injecting arguments were too hard and time consuming to be implemented. Therefore we let this work as our future work, and for now we assume that our test functions always have such array argument for DRAM interface as follows:

```
int top(int dram[256], /* func args */) {
  ...
}
```

which can be fed to our tracer record function calls by using `llvm::Function::getArg()` function.

3.6 DRAM Buffer Wrapping

Although DRAM is larger than BRAM, it is still a finite resource so that is not able to contain every trace data. To guarantee error-free implementation, we make tracer's `record` function include buffer wrap: once all buffers are filled, the function will overwrite the entry at the first index, removing the oldest trace.

To indicate whether it is wrapped or not and which data is the first one, we use two 32bit integers at the end of the DRAM buffer. To mitigate the overhead of writing such information, these two integers are only written to DRAM in `finish` tracer function.

```
void record(int *array, ...) {
  /* write row and col */
  current_index += 2;
  wrapped |= (current_index & wrap_mask);
  current_index &= (wrap_mask-1);
}

void finish(int *array) {
  array[buffer_size - 2] = current_index;
  array[buffer_size - 1] = wrapped? 1 : 0;
}
```

where `wrap_mask` equals to buffer size - 2. For instance, if buffer size is 258 (2^8+2), `wrap_mask` is 256 and actual data will never be written above 256th index.

Once program terminates, we could sort trace data in order using `wrapped` and `current_index` as illustrated in Algorithm 2. If buffer is wrapped, data at `current_index` is the oldest one, hence it has to be added first for sorting data in order.

Algorithm 2 Trace data sort algorithm.

```
1: trace := ∅
2: if wrapped == 1 then
3:     for i := current_index ∼ buffer_size do
4:         trace ← array[i]
5:     end for
6: end if
7: for i := 0 ∼ current_index do
8:     trace ← array[i]
9: end for
```

4 Evaluation

4.1 Evaluation Setup

The control flow tracer is evaluated with post-synthesis co-simulation of the synthesized RTL with the original C/C++-based test bench, using the Vivado simulator (XSIM) provided by Vitis. For the scope of this work, where our interest is in synthesizing C/C++ functions into Verilog, verifying the functional correctness of the generated Verilog with co-simulation would suffice to evaluate our tracer. Though co-simulation does not include a model for an actual DRAM, a FIFO verification adapter is attached to the AXI interface in its place. Since the synthesized HLS design and the DRAM controller will interface via the AXI4 protocol, this setup suffices to verify read/writes to the DRAM. Timing constraints are expected to change, however, when we attach an actual DRAM to the AXI interface to use the framework in an FPGA.

As there are no known FPGA benchmarks targeting C/RTL co-simulation, a total of six synthetic test cases were created to evaluate the control flow tracer with varying operations. Test cases range from a simple for-loop accumulating the sum of values in a variable, to more complex cases including nested for, while, and do while loops, as well as loops inside conditional statements. One of the test cases involves calls to another function from the top-level function. Table 2 summarizes the workload characteristics of each test case.

4.2 Evaluation Results

4.2.1 Latency Overhead

As evident in table 3, measurements of the increase in average latency from utilizing the control flow tracer shows that the tracer adds a non-negligible overhead to the user module. The increase in execution time is not only due to the tracing process itself, but also the fact that injecting the tracer prevented Vitis from doing various optimizations, such as automatic loop unrolling. While Vitis could fully unroll loops in the user module when

Test Case	Workload Characteristics
sigma	Single loop
sigma_looped	Loop inside conditional, conditional inside loop
nested	Nested loop
loop	Loop inside conditional, conditional inside loop
hotloop	Loop inside conditional
call	Conditional inside loop, function calls

Table 2: A description of each test case and its workload characteristics.

Test Case	Before	After	Factor
sigma	61	1575	25.82
sigma_looped	59	13856	234.85
nested	29	7504	258.76
loop	64	2858	44.66
hotloop	131	6867	52.42
call	29	3568	123.03

Table 3: Average latency before and after tracer injection.

	Before		After		Factor	
Test Case	FF	LUT	FF	LUT	FF	LUT
sigma	565	567	2080	3268	3.68	5.76
sigma_looped	567	567	2757	6057	4.86	10.68
nested	143	243	1333	3012	9.32	12.40
loop	605	631	2669	5208	4.41	8.25
hotloop	229	504	1398	3123	6.10	6.20
call	143	243	1525	3838	10.66	15.79

Table 4: Flip flop (FF) and lookup table (LUT) usage in top-level function before and after tracer injection.

executing without our instrumentation, adding the tracer lead to pipelined loops with initiation intervals as large as 144 clock cycles.

4.2.2 Resource Overhead

In terms of resource usage, utilizing the tracer adds a constant overhead of 133 flip-flops and 452 lookup tables to implement the `finish()` function. The `record()` function adds on average 388 flip-flops and 831 lookup tables, with slight variances incurred from Vitis pipelining the function. Table 4 illustrates the number of flip flops and lookup tables used to implement the top level function, in the original user module and with the tracer functions injected. Adding control flow tracing has limited Vitis from performing optimizations to reduce the area of the synthesized design.

4.3 Loop Unroll Factor Exploration

In this section, we suggest and demonstrate a compelling use case of our tracing framework: Automatically exploring the resource efficiency of various loop unroll factors.

When designing hardware with HLS, developers have the choice to unroll a loop to trade off resource usage for higher performance. This decision has two parts: *which* loop to unroll and by *what unroll factor*. However, neither decisions are usually based on actual usage data - a fundamental lack in hardware observability as we argue throughout the paper. Thus developers are forced to make guesses when deciding.

With control flow tracing enabled, the developer now has access to multiple traces each corresponding to one invocation of the circuit on actual production data. With this, the developer has concrete reason to find loops that are most frequently executed, i.e. the ones that are likely to yield more performance benefit when unrolled.

We demonstrate this scenario With the `hotloop` test case (see Table 2). First, we collect multiple trace data by invoking the synthesized circuit on multiple different input data. Second, we analyze the trace to identify the hottest loop, i.e. the loop that has been most frequently executed across all traces. Then, we explore 6 different loop unroll factors for the hottest loop and plot the change in resource usage, simulated execution latency, and resource efficiency (see Figure 3). Given trace data, all subsequent procedures are automated.

As can be seen even from this simple test case, the relationship between unroll factor and resource usage/efficiency is highly non-trivial. As the loop unroll factor increases, more look-up tables are used, but flip-

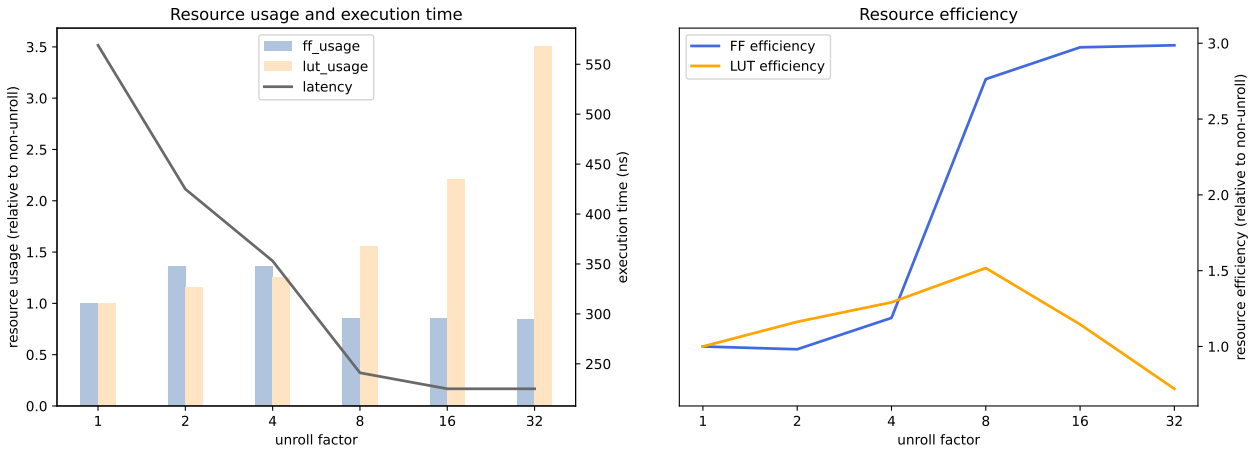

Figure 3: Resource usage, execution time, and resource efficiency from unroll factor exploration.

flops actually end up being used less.[1] Given the control flow trace and Figure 3, the hardware developer can make more informed choices for *which* loop to unroll and *what unroll factor* would fit their use case.

5 Possible Extensions

In this section, we present some ideas about extending the HLS tracer framework and how one might implement them.

5.1 Recording Timestamps

Currently, the trace data collected by the HLS tracer are the line and column number of the trace points. On top of this, collecting the timestamp of each trace point is will provide significant benefit in terms of optimizing and debugging performance. Especially, performance debugging will benefit from timestamps because there are certain operations written in high-level language that incur unexpected overhead. For instance, the developer may casually declare and initialize a local array like this:

```
void top() {
  int array[1024] = {0,};

  // Use array.
}
```

However, this code silently creates a loop that assigns zero to each element, thereby incurring 1024 cycles of latency overhead by default.[2] This overhead is undue when all array elements are never read before being written at least once. This kind of unexpected overhead in terms of cycles will become evident when the current cycle count is also collected as part of the trace, acting as a timestamp.

Yet, in Vitis HLS, it is currently not possible to read the clock signal wire (`ap_clk`) in high-level code. Moreover, HLS synthesis deals with single threaded execution, thus not allowing some child thread to count the number of cycles in parallel with the main logic of the user module. Thus, it is imperative that the cycle-counting module be implemented as a separate module from the user module. The cycle-counting module can be programmed in HLS like this:

```
void top(volatile long long *clk) {
  *clk = 0;
  while (1) {
  #pragma HLS pipeline II=1
    (*clk)++;
  }
}
```

The signal `clk` can be connected to the user module through the top-level function's input arguments, and fed into the `record` function.

5.2 Trace Caching in BRAM

The current implementation of HLS tracer writes data to the trace array immediately when possible, initiating

[1] We believe this is because the internal logic (and hence the truth table) becomes more and more complex, forcing the backend to implement it with powerful look-up tables. However, analysis of this phenomenon is not the central focus of this work.

[2] There is an upper limit in the number of concurrent writes to BRAM. Thus, when local arrays are mapped to BRAM, it may not be possible to fully unroll this initialization loop, making latency overhead inevitable. Array partitioning may be considered, but this in turn has the possibility of incurring resource overhead due to the minimum granularity of BRAM slices.

an AXI write operation every time. This incurs its due latency, having negative effects on performance factors such as loop pipelining initiation intervals, as delineated in section 4.

One way to mitigate this problem would be to allocate an acceptable amount of BRAM on the FPGA as a cache for trace data. Once the amount of cached data reaches a certain threshold, e.g. when the allocated BRAM is full, everything is flushed through the AXI interface at once. Contrary to when trace data is exported one by one, flushing a large amount of data can leverage the AXI burst feature. Specifically, the AXI interface can be set to burst mode and the initial offset and the length of the burst is provided. Burst write incurs less total latency compared to initiating writes one by one.

Note that more than one burst may be required since the amount of cached data may exceed the amount of space left in the trace array in DRAM, thereby incurring a wrap. Since burst mode can only be used for arithmetically increasing address sequences, two bursts must be made. In addition, the number of burst operations needed and the length of each burst must also be determined, increasing the logic complexity of the tracer itself.

6 Conclusion

In this work, we designed and implemented the HLS tracer framework, an extension of Vitis HLS that transparently instruments the user's high-level source code to allow in-hardware control flow trace collection. We also show the practical usefulness of our framework through a concrete case study.

However, our work is also limited in some respects. First, our tracer framework adds non-negligible overhead to the user module. Optimization opportunities, such as BRAM trace caching (section 5.2), are left as future work. Second, our framework does not guarantee with cosimulation that it will satisfy DRAM timing constraints. That is, our framework only creates and tests up to the AXI interface, and cosimulation merely attaches a FIFO verification adaptor to the AXI interface without precisely modelling the timing constraints of actual DRAMs. One future work will be to create a model for DRAM with timing parameters and include that in the cosimulation target.

Despite its limitations, our work is the first open implementation of control flow tracing for HLS, and we believe that the step it undertook is a significant one. Trace collection will facilitate research and development in not only profile-guided optimization, but also convenient debugging for HLS. Especially, the fact that our trace is collected directly on hardware will bring the HLS developer one step closer to actual hardware where their synthesized modules run.

References

[1] CANIS, A., CHOI, J., ALDHAM, M., ZHANG, V., KAMMOONA, A., ANDERSON, J. H., BROWN, S., AND CZAJKOWSKI, T. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (2011), pp. 33–36.

[2] CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., AND HWU, W.-M. W. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience 22*, 5 (1992), 349–369.

[3] COUSSY, P., GAJSKI, D. D., MEREDITH, M., AND TAKACH, A. An introduction to high-level synthesis. *IEEE Design & Test of Computers 26*, 4 (2009), 8–17.

[4] GAJSKI, D. D., DUTT, N. D., WU, A. C., AND LIN, S. Y. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[5] GOOGLE. Xls. https://github.com/google/xls, 2020.

[6] GUPTA, R., MEHOFER, E., AND ZHANG, Y. Profile guided compiler optimizations.

[7] HUNG, E., AND WILTON, S. J. Towards simulator-like observability for fpgas: a virtual overlay network for trace-buffers. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays* (2013), pp. 19–28.

[8] KATHAIL, V. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2020), pp. 173–174.

[9] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[10] MONSON, J. S., AND HUTCHINGS, B. L. Enhancing debug observability for hls-based fpga circuits through source-to-source compilation. *Journal of Parallel and Distributed Computing 117* (2018), 148–160.

[11] NANE, R., SIMA, V.-M., PILATO, C., CHOI, J., FORT, B., CANIS, A., CHEN, Y. T., HSIAO, H., BROWN, S., FERRANDI, F., ET AL. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35*, 10 (2015), 1591–1604.

[12] PILATO, C., AND FERRANDI, F. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications* (2013), IEEE, pp. 1–4.

[13] WHEELER, T., GRAHAM, P., NELSON, B., AND HUTCHINGS, B. Using design-level scan to improve fpga design observability and controllability for functional verification. In *International Conference on Field Programmable Logic and Applications* (2001), Springer, pp. 483–492.

[14] WINTERMEYER, P., APOSTOLAKI, M., DIETMÜLLER, A., AND VANBEVER, L. P2go: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (2020), pp. 146–152.

[15] XILINX. AXI Master Interface. https://www.xilinx.com/html_docs/xilinx2020_1/hls-guidance/qoa1585574520885.html, Accessed on Dec 8, 2021, 2021.