Automated Bug Detection for Pointers and Memory Accesses in High-Level Synthesis Compilers

Pietro Fezzardi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
email: pietro.fezzardi@polimi.it

Fabrizio Ferrandi
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy
email: fabrizio.ferrandi@polimi.it

***Abstract*—Modern High-Level Synthesis (HLS) compilers aggressively optimize memory architectures. Bugs involving memory accesses are hard to detect, especially if they are inserted in the compilation process. We present an approach to isolate automatically memory bugs introduced by HLS tools, without user interaction, using only the original high-level specification. This is possible by tracing memory accesses in software (SW) and hardware (HW) executions on a given input dataset. The execution traces are compared performing a context-aware HW/SW address translation, leveraging alias-analysis, HLS memory allocation information and SW memory debugging practices. No restrictions are imposed on memory optimizations. We show results on the relevance of the problem, the coverage, the detected bugs. We also show that the approach can be adapted to different commercial and academic HLS tools.**

I. Introduction

The complexity of hardware designs is constantly increasing. To manage this growth and shorten development cycles, High-Level Synthesis (HLS) is steadily becoming more popular. HLS tools are getting more sophisticated, as new algorithms are developed for scheduling, module allocation, resource sharing, memory allocation and other tasks. Often, these algorithms consider more than one of these problems at the same time. Equally often, the more complex and advanced is the algorithm, the more dissimilar is the final HW from the original high-level specification, and the harder to debug when the HLS implementation is wrong. In this paper we focus on memory allocation. In particular, we discuss a methodology for debugging memory faults introduced by HLS tools.

Optimizing memory allocation is very important in HLS. Several different results show that it leads to significant improvements of the generated designs [1] [2] [3] [4] [5] [6] [7]. Every technique has benefits and subtleties. Thus, a flexible methodology for debugging memory allocation implementations in HLS tools must be kept independent from the underlying memory technologies and allocation algorithms.

A consequence of having so many options for memory, is that the resulting Hardware Description Language (HDL) can be hard to trace back to the original high-level source code. Synthesis tools usually rely on naming conventions to preserve variable names across abstraction levels, to simplify design verification. Beside that, in HLS it is necessary to correlate runtime assignments of variables in SW with signal variation happening in HW. Things get more complicated when the starting language for HLS is `C`. Pointers, access-by-address, and dynamic allocation (with `malloc`/`free`) complicates memory synthesis. Despite these obstacles, the large majority of academic and commercial HLS compilers are based on `C`. In fact, a number of improvements have been made and are currently ongoing to overcome the troubles with the mentioned features of the language [8]. For this reason, a debugging technique for HLS of memory subsystems must be able to handle the quirks of `C` to be both effective and useful.

Recently, there have been efforts to endow HLS frameworks with tools for automated analysis of the discrepancies between the executions of high-level source code and generated HW. The one described in [9] relies on compiler information generated by the *LegUp* HLS compiler [10]. Another work [11] extends the idea to support compiler optimizations and temporary variables, giving also a formal description of the algorithm. However, none of the approaches described in these works can be applied to debug memory allocation and pointer arithmetic. In [11], the impossibility to compare SW pointers and HW addresses is explicitly mentioned as the primary cause of the limited coverage of the method.

Finding bugs in how HLS tools handle memory is an important problem if HLS is going to achieve more widespread adoption. Methodologies and utilities to find corner cases where HLS tools do not work correctly are essential to debugging them and to making them more stable, reliable, and robust for production use.

In this paper we specifically address this problem. In particular, we describe how to use HLS information to create a HW/SW Address Space Translation Scheme (ASTS). Its definition is kept as general as possible, to support the widest variety of memory architectures and optimizations. The ASTS enables the discrepancy analysis of pointer operations, mapping SW pointers onto HW memory locations. As with normal discrepancy analysis, the approach detects bugs as soon as they originate. This is useful, because it makes possible to isolate the failing operation and the transformation that introduces it, along with a number of information to back-track the bug to the high-level source code.

The remainder of this work is structured as follows: in section II we describe some preliminary concepts and assump-

tions necessary for the description of the ASTS; in section III we formally introduce the ASTS; in section IV we show how the ASTS works in practice and how it can be used for address discrepancy analysis; in section V we report data collected on different sets of benchmarks, to demonstrate the big impact that memory and pointer operations have on memory intensive applications, and to show that other tools can benefit from the approach; finally, in section VI, we summarize the results, outlining possible future research.

II. Background and Assumptions

HLS tools use different techniques to synthesize pointers in HW. For our analysis we evaluated three different HLS compilers, with support for `C` pointers: 1) *LegUp* [10], based on LLVM [12] and developed at the University of Toronto; 2) *bambu*, based on GCC [13] and developed at Politecnico di Milano [14]; 3) a recent version of a production-ready commercial HLS tool targeting Xilinx FPGAs, referred in the following as *CTool* (the license does not allow to disclose the name). All of them are able to synthesize pointers, with different memory partitioning and allocation schemes. *LegUp* and *CTool* do not support dynamic memory allocation, while *bambu* does, through the synthesis of `malloc` and `free`. To understand the design of the Address Space Translation Scheme (ASTS), it is necessary to dig deeper in how pointers and memory accesses are mapped to HW. Obviously, every single HLS tool has its own Intermediate Representation (IR), and performs different analyses and optimizations. The evaluation of the three tools was necessary to find a general model able to describe their operation at a higher level. The methodology we introduce should be flexible enough for different HLS frameworks.

A. Alias-Analysis and Pointer Synthesis

In general, for memory allocation, HLS tools take two decisions: 1) which variables have to be stored in memory (usually global, `static`, `volatile`, arrays, and `structs`, but possibly others); 2) the location where every memory-mapped variable is stored (with different partitioning schemes). The first point is usually inferred using *alias-analysis* (or *points-to analysis* [15]) and/or decided with explicit directives. The process is described in [8], which also outlines some optimizations to save resources and superfluous accesses. The second option is tightly bound to the HLS implementation, to the specific memory architecture of the generated design, and to the set of performed memory optimizations.

For every given combination of these things, we can make a general assumption: every variable must be mapped to a specific memory location. A *memory location* is an unambiguous position in the generated HW. It can be described with a unique identifier for the memory module, a position in that memory module and the size of the object stored in that position. This holds for any underlying memory technology, being it a ROM, a BRAM, or an external DDR. With this assumption we can state that every memory-mapped variable i is associated to a memory location. We define a memory location as a triplet $\langle M_i, B_i, S_i \rangle$, where M_i is a unique identifier for a memory module (independent of the memory technology), B_i is an offset in bytes in that memory module, and S_i is a size. In general, the size S_i must be expressed in multiples of the memory alignment. In the following, we assume byte alignment, but any other alignment can be used without loss of generality. This concept of memory location is similar to the *location sets* introduced by Wilson and Lam [16] and also used by Séméria and De Micheli [8]. In compilers, alias-analysis works on pointers, trying to recover the set of memory locations where they can point to: the *points-to set*, or *location set*. The results of the analysis can be used to tune the HW memory partitioning. Notice that the points-to sets, like our memory locations, are abstract concepts, independent of the target architecture. The authors of [8] describe how to use them to enhance the address decoding circuits. They use *tags* and *indexes* as internal encoding to optimize pointer operations in HW, getting rid of tags or indexes when possible. In this respect, the HLS compilers we evaluated adopt different IRs and map addresses to HW with their own conventions, but their address encoding is equivalent to [8]. The only thing necessary for the implementation of the address discrepancy analysis is that, using HLS information, it has to be possible to identify the signals representing the addresses in HW. The values of those signals will be used by the address discrepancy analysis algorithm. Like in [8], M_i is an abstract identifier, that may actually not be translated in HW, depending on optimizations and static analysis. In particular, when a memory operation can be attributed to a single local/private memory module, M_i can be completely optimized away. In cases where M_i is not translated to HW, only B_i and S_i can be used for the discrepancy analysis. When B_i and S_i are optimized away, M_i is sufficient to retrieve the memory location.

The proposed approach aims at handling complex memory allocation patterns. To support array partitioning, the assumption must be slightly restricted. Requiring a variable to be mapped on a single memory location is clearly not enough. Instead, we require that *every element* of the array itself (or field in a `struct`) is associated with a single memory location. To summarize, this means to be able to compute the inverse function of the mapping of high-level variables onto HW memory locations. In case of arrays and `structs` this inverse function needs to have per-element granularity. To avoid to weigh down the terminology, in the following the term 'variable' is used loosely, with the meaning of 'scalar variable or field in a `struct` or element of an array'. In this way we can keep the discussion general while sketching the approach, but treating all the data types in the same way.

B. Additional Assumptions

Before diving into the description of how address and pointer operations can be automatically checked for discrepancies, it is necessary to describe the kind of debugging flow where this technique can be employed. Fig. 1 depicts a general discrepancy analysis debugging flow, very much akin to what is described in [11]. We are now interested only in the parts

with blue background, representing the standard discrepancy analysis debug flow. The parts with green background are necessary for the analysis of pointers. They are introduced by this work and described in detail in sections III and IV.

The top section of the figure is the classic HLS flow. The portion marked "EXTRACTED INFORMATION" shows the data extracted from HLS that are necessary for automated debugging with discrepancy analysis. Some of this information is also fed back into the HLS process itself. The area marked "DEBUGGING STEPS" contains the data and the operations directly involved in the analysis of the discrepancies. On the left, the instrumented high-level source code in Static Single Assignment form (SSA) is compiled and executed to generate the SW traces. On the right, the generated HDL is simulated with cycle accuracy, to produce the HW traces. In the middle, the proper analysis of the discrepancies takes place, using HW and SW traces. The analysis produces a number of useful results, shown on the bottom of the figure.

This kind of analysis is not in the family of equivalence checking, because it cannot guarantee formal equivalence between the high-level source code and the generated HW. Rather, for a given input set, it extends the granularity of functional verification to find bugs at every level in the HW hierarchy. The discrepancy analysis works on predetermined input data sets and uses them to automatically generate a rich and fine-grained set of runtime assertions, to detect the first place where HW and SW execution mismatch. However there are two big differences between discrepancy analysis and traditional assertion-based verification [17] [18] [19]. The first is that with discrepancy analysis the assertions are generated automatically from the high level source code and the input dataset, without user action. The second is that they are not translated into assertion checker circuits to be embedded in the design. Instead, they are translated into informations to build HW and SW execution traces, that are collected and analyzed off line to detect mismatches.

The discrepancy analysis flow depicted in blue in Fig. 1 is known not to be able to handle discrepancies on addresses and pointers. Thus, it is necessary to find a way to map SW addresses to HW addresses: the *Address Space Translation Scheme* (ASTS). The approach described in this work builds on top of the flow depicted in the picture, slightly altering some of the involved steps to produce the data necessary for the correct handling of addresses and pointers. Extending this flow to support pointers allows us to keep all the benefits of the approach. First of all, it is possible to maintain information on the relationships between original source code and generated HDL [9]. The same holds for signals and registers representing temporary variables inserted by the compiler for optimizations [11] [20]. Using similar techniques, we are able to select the HW signals necessary to track operations involving C pointers and address arithmetic. The result of every one of such operations is an address, which represents a HW memory location, as described above. The selection is completely machine-driven, without user intervention. In the same way, using scheduling information, it is possible

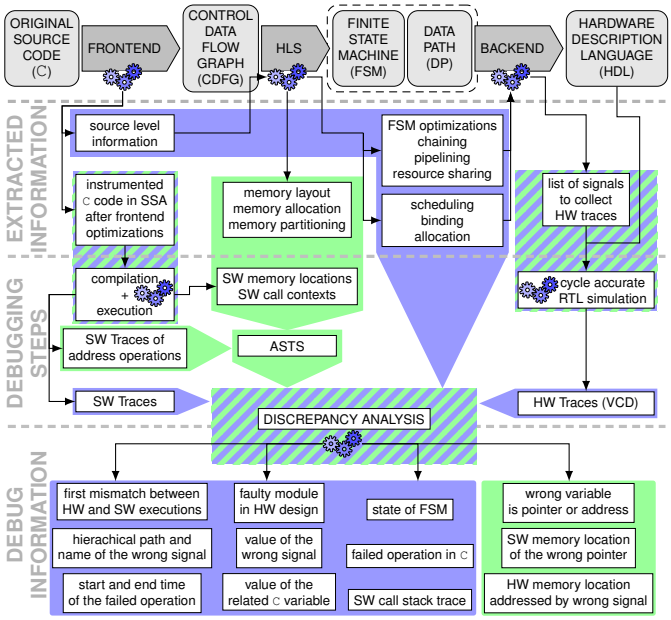

Fig. 1: Representation of the debugging flow based on automated discrepancy analysis. The top section of the picture is the typical HLS flow. Everything below it represents data and execution steps necessary for discrepancy analysis. Plain boxes are the data, while boxes with gears are execution steps. Steps and data with blue background are part of the standard discrepancy analysis flow. Parts with green background are necessary for address discrepancy analysis, and are introduced in this work. Parts with mixed blue-and-green background are present in the normal flow, but they must be extended and modified to enable address discrepancy analysis.

to detect the states of the HW Finite State Machine (FSM) when these signals are written. After the signal selection, it is possible to retrieve the traces of their variations during HW execution. This can be done on chip [9] [20] or relying on simulation [9] [11]. In this work, simulation is used to obtain complete visibility of the signals, independently of the memory architecture and the underlying technology. If the observability of the signals can be achieved on-chip without altering the memory architecture, it should be possible to use our approach also for in-system debug. On the other hand, the execution traces of the software can be obtained adding instrumentations to the original C source code. Then the code is compiled and executed with the additional instrumentations, the traces are dumped to file and they can be easily collected and analyzed with the approach described in [11]. Clearly the SW traces described there are not enough to compare addresses. Hence they must be enriched with some information, necessary for the construction of the ASTS. However, the approach for their generation remains the same. Section III describes what is necessary to build the ASTS.

These assumptions give us all what we need to proceed. We have a way to retrieve execution traces from HW and SW. We also have a way to relate pointer operations in C to HW addresses. However, even if in our model HW memory locations are easy to transform in addresses, HW traces are not easy to compare directly with the corresponding SW traces. The reason is the big difference between HW and SW address spaces. In SW we have a single contiguous memory. Even if it is actually divided in different segments (stack, heap, static),

the raw binary values of a pointers are always directly comparable and it is easy to see if they are equal. If two pointers are equal in SW they must point to the same memory location. On the other hand, in HW, the thing more similar to pointers are signals representing addresses in memory modules. However, memory in HW can be fragmented, spread across different memory modules, possibly mapped on different technologies. Hence it is totally possible to have two different signals, both used for addressing memory modules, that have at the same time the same value but are actually used to access different memory locations. This happens when they are bitwise equal, but they are used for addressing different memory modules. This poses a major challenge to the verification of memory subsystems generated with HLS, in particular when aggressive optimizations of the address decoding logic are performed.

C. HLS Flow Error Coverage Metric

We define a static coverage metric, to measure the effectiveness of the address discrepancy analysis. The metric is called *instruction coverage* and it has some analogies with statement coverage for a C program. It is denoted with icov:

$$\text{icov} = \frac{\text{\# of checked static operations}}{\text{\# of static operations}}$$

It measures the percentage of static operations in the elaborated C program that can be checked with the address discrepancy analysis. At first sight it may resemble *statement coverage*, which in general is defined as follows:

$$\text{scov} = \frac{\text{\# of statements executed at least once at runtime}}{\text{\# of static statements}}.$$

In the remainder of the paper we will use the notation ccov for C statement coverage and vcov for Verilog statement coverage.

Despite the similarities, *instruction coverage* is not equivalent to the *statement coverage* in the original C code nor in the generated HDL. First of all icov is a static metric. In general, *statement coverage* is dynamic, because it measures the number of statement that are actually executed at runtime. Instead, icov measures at compile time the number of instructions that can be checked, even if at runtime such instruction are not executed, depending on the test input. Secondly, icov has a finer granularity, because it considers the values of the intermediate subexpressions in statements separately. In this way, it is possible to check variable assignments, but also intermediate values assigned in composite statements. This metric measures the granularity of the checks. The goal is to show that a large part of the operations are directly checked for discrepancies, even if some cases are inevitably lost.

Clearly the *instruction coverage* cannot be full if there are some control flow instructions. Branch statements, function calls and return statements, cannot be checked directly by the discrepancy analysis, because they do not assign variables. However, they can be checked indirectly by the discrepancy analysis. In fact, function calls are not directly checked, but the instructions in the body of the called functions are. Return statements are not directly checked inside the body of the returning function, but the returned values are checked right after their evaluation, before the return. Finally, branch instructions are not directly checked, but the branch condition is checked at its evaluation, before the jump. The key is that the checks enabled with discrepancy analysis are on the assigned values. While calls, returns and branches do not assign variables, they alter the control flow. Control flow mismatch can be detected with the approach described in [11]. Results reported in section V-D show a good coverage even excluding the mentioned cases.

III. Definition of the ASTS

In the HLS process, memory allocation has to decide for every variable i a HW memory location $\langle M_i, B_i, S_i \rangle$ and to instantiate the necessary memory modules. The notation is the same used in section II-A. In software, the same variable i will be allocated at runtime at a certain memory location, not known at compile time. This memory location can be represent in a shorter form for SW: $\langle CB_i, CS_i \rangle$. Here CB_i is simply the address of the variable and CS_i its size. However, in software, the memory location where a variable is allocated at runtime, does not depend only from the identifier i alone. Consider a program where the `main()` calls another function `fun()` multiple times, and `fun()` has a local stack-allocated variable accessed by address. In this case, at every call to `fun()`, the SW memory location for the variable will be different. On the other hand, multiple strategies can be adopted from the HLS engine for the synthesis: `fun()` could be inlined; a separate module for `fun()` could be generated; the module used for the implementation of `fun()` could be duplicated and instantiated once for every different call site. Depending on this decision the HW memory location for the local variables of `fun()` may vary. However, this decision must be taken during the HLS allocation step, and it is fixed at the end of the HLS flow itself. In SW, instead, information on the call context is necessary to handle this case, because SW memory locations are dynamically determined at runtime and depend on the call context. To extract runtime information from SW, we insert additional memory profiling instrumentation in the C code, before it is compiled for trace generation. In this way it is possible to dump the SW memory mapping at runtime and to extract the data necessary to build the ASTS. In particular, for every function call we generate and collect the unique context id j. For every memory-mapped variable we dump the identifier i and its SW memory location, composed by its base address CB_i and size CS_i. To support array partitioning across memory modules, this must be refined. In particular, i must be a unique identifier for an element in the array (or field in a `struct`).

Starting from this information, it is possible to define the *Address Space Translation Scheme*. The ASTS, is subdivided in two tables, which can be implemented efficiently as hash tables. The first is called *Software Address Table* (SAT) and contains data on SW memory locations. One row is in the form $[j, \langle CB_i, CS_i \rangle, i]$. Here j is the primary key and $\langle CB_i, CS_i \rangle$ is the secondary key. They allow, for any given call context, to retrieve efficiently the mapping of SW addresses on variables.

Algorithm 1: Address Discrepancy Analysis Routine

```
   Shared Data: ASTS = (SAT, HAT)
1  bool discrepancy (j, s, h)
       Input  : j: context identifier
                s: SW address assigned to a pointer p in j
                h: value of the signal related to p in HW
       Result : true if s and h mismatch, false otherwise
2      i = search (j, s) in SAT;
3      if (i is found) then
4          ⟨M_i, B_i, S_i⟩ = search(i) in HAT;
5          if (⟨M_i, B_i, S_i⟩ is found) then
6              h' = decodeHW (⟨M_i, B_i, S_i⟩);
7              if h ≠ h' then
8                  return true;
9              else
10                 return false;
11         else
12             // i is not allocated in memory in HW
13             return true;
14     else
15         // s is not in range for any variable
16         return false;
17 decodeHW (⟨M, B, S⟩)
       Input  : An HW memory location
       Output : Value in HW associated to the base address
                of the input memory location
```

This means that for every given call context j and for every pointer p in that context, a fast lookup in the SAT is enough to determine the variable i where p points. The second table is the *Hardware Address Table* (HAT). It is composed by the fields $[i, \langle M_i, B_i, S_i\rangle]$. The HAT is build during the memory allocation step of the HLS process. It maps every variable i to a HW memory location $\langle M_i, B_i, S_i\rangle$.

An important thing for both HW and SW memory locations is that even if they are expressed in this formal notation, they are easily convertible to bit sequences and back to memory locations. For SW this is trivial, given that CB_i is actually an address. For HW, the mechanism strictly depends on the implementation, but it is necessarily computed during HLS for memory allocation and to build the address decoding logic.

Another important consideration is that the construction of these tables happens after HLS, so that all the compiler optimizations have already finished. This is particularly important for memory to register and register to memory transformation passes that could alter the construction of the ASTS. Building the tables after HLS allows to treat also variables that have been moved from memory to registers and vice versa.

IV. Debugging Memory Allocation

A. Address Discrepancy Algorithm

At this point, HW traces, SW traces and the ASTS are glued together with an algorithm for automated discrepancy detection for pointer operations. The HW traces consist of Value Change Dump (VCD) files [21]. The SW traces represents the values of every assignment of a variable during SW execution. For normal variables, the discrepancy analysis works just like in [9] and [11], with bit-per-bit comparison. For pointers, bit-per-bit comparison would obviously lead to a mismatch, even if the synthesized address decoding logic is correct, because of the different address spaces. For this reason the match is evaluated with Algorithm 1.

The function `discrepancy` describes the algorithm at a high-level. It takes 3 arguments: j is the SW context identifier; s is an integer representing a SW address assigned to a pointer variable p in context j; h is the value of the signal representing in HW the pointer p. The function initially performs a lookup in the SAT to compute the variable i pointed to by the address s (line 2). If the lookup fails, it means that the address is not in range for any variable in SW. This means that s does not point to any valid SW memory location. Hence the discrepancy analysis cannot give conclusive results because there is no SW memory location to compare with HW (lines 14-16). If the lookup in the SAT succeeds, then the variable i is used as key for a second lookup, this time in the HAT (line 3-4). If this second lookup fails, it means that there is a variable i in SW whose address is taken but that is not mapped to memory in HW. In this case the function returns an error (lines 11-13). Instead, if also this second lookup succeeds, the computed HW memory location $\langle M_i, B_i, S_i\rangle$ is converted to an integer with the `decodeHW` function (line 5-6). The decoded value h' represents the HW address expected to match the SW address s computed in SW. Thus, if h≠h', a mismatch is detected otherwise `discrepancy` returns false.

The `decodeHW` is strictly dependent on the implementation. Hence it is different for every HLS tool, since it uses a lot of HLS information on memory allocation, and on how HW addresses are actually mapped to HW. Some insight will be given in section V-B, where we will describe briefly how the `decodeHW` can be constructed for the tools we evaluated.

B. Refining Address Discrepancy Analysis

The presented approach has to be refined to avoid false positives. The first class of such false positives happens when the synthesized HW performs a speculated READ. This is perfectly possible in HW, but in SW it may access an invalid address, causing segmentation fault. Hence we must ensure to avoid READ speculation. This is not really a problem, since none of the evaluated HLS tools is able to perform it.

Other problems arise when the points-to set for a given address contains two arrays contiguously allocated in memory by the `C` code. An example is the code in Fig. 2, where `a` and `b` are contiguous. *After* the last iteration of the first loop, `p` points to `b[0]`, thus it is in-range for `b`. The reason is that, *after* the last iteration of the first loop, `p` is set to `&a[32]`, which causes the loop to end. This assignment is actually performed *before* entering in the second loop and setting `p` to `b`. This means that there is a time when `p` evaluates to `&a[32]`, which in `C` overlaps with `&b[0]` but in HW it may not. This is not a problem in `C`, but in HW `a` and `b` could even be mapped on different memory modules. At this point, if the value of `p` in SW is compared with HW, it is likely to generate a false positive. A possible solution is to insert poisoned

```
extern int something(int *p);
int main() {
   int *p, a[32], b[32], res = 0;
   for (p = a; p < a + 32; p++)
      res += something(p);
   for (p = b; p < a + 32; p++)
      res += something(p);
   return res;
}
```

Fig. 2: A C program causing a false positive

```
int w(struct sockq *q, void *src, int len) {
    char *sptr = src;
    while (len--) {
        q->buf[q->head++] = *src++;
        if (q->head == NET_SKBUF_SIZE)
            q->head = 0;
   }
   return len;
}
```

Fig. 3: A C function with pointer operations not supported by *CTool*.

redzones between different memory-allocated areas in C, using the AddressSanitizer (ASAN) memory error detector [22], deployed in both GCC and LLVM. The C code used for the discrepancy analysis is compiled using ASAN. ASAN consists of a compiler instrumentation pass and a run-time library which replaces the `malloc` function. Using ASAN results in two advantages: it avoids memory bugs in the original high-level code used to generate the traces; it avoids false positives cause by contiguously allocated data. Algorithm 1 cannot really say anything about out-of-range addresses, because only in-range addresses are actually used for lookups in the HAT. ASAN is a complementary solution to this problem: mismatches for out-of-range addresses are not reported, but ASAN ensures that there are no dereferences. Another option would be to perform static range checking directly in the HLS tool, which is partially done by most compilers with the correct flags. However, static range checking is not always exhaustive in all the cases. Hence, the instrumented code for trace generation would still need to be compiled with a dynamic range-checking library. ASAN solves all these issues at once and it is guaranteed to improve with time, following the development of mainstream compilers.

V. Experimental Results

A. Experimental setup

The approach have proved to be versatile and effective in automatic bug detection on different designs, synthesized with different HLS tools. We used two groups of benchmarks: the CHStone HLS benchmark suite [23] and the GCC C-torture test suite [24]. The first establishes a common baseline for all the tools, but it has a big drawback for the scope of the research: it has no complex pointer operations. They are complicated enough to make alias-analysis non trivial, but they really don't try to push the limits of what can be done with pointers in valid C code. This is not useful when testing the address discrepancy analysis, because the focus is on seeing how it behaves with exotic pointers manipulation in C. This is the main reason behind the decision to use also the GCC C-torture tests. These are a large set of self-contained C programs, specifically designed to exercise corner cases of a standard-compliant C compiler, including a number of uncommon things with pointers. From preliminary trials, it turned out that only 216 of such tests were involving pointer operations. The analysis is restricted to this subset. On 56 of them, *CTool* failed to complete the HLS process. An example of a C code snippet that could not be handled is shown in Fig. 3. Notice that from the programmer standpoint the use of pointers in this function it is not particularly strange. This kind of syntax is common practice in embedded C code, but still many commercial tools have problems in handling pointer casts and other similar operations. For this reason, for the implementation of a proof-of-concept debugger with automated discrepancy analysis, we choose *bambu*, which also has an advanced approach to memory allocation [2] and it implements fairly complex frontend optimizations [14]. Fig. 1 depicts the discrepancy analysis flow augmented with the steps necessary for the creation and use of the ASTS. For the experiments we used GCC 5 with ASAN to compile the sources to obtain SW traces. The simulation used to generate HW traces is cycle-accurate and it has been performed with ModelSim SE-64 10.3 from Mentor Graphics.

B. Applicability to Other Tools

In an effort to remain as general as possible, despite our choice of *bambu* for the actual implementation, we tried to evaluate the method we propose with all the tools we considered for this study. The main obstacle to this evaluation is that, as described in section III, some of the information for the construction of the ASTS must be extracted from the HLS compiler. This is also necessary to understand how to design the implementation-dependent `decodeHW` function. For *bambu* and *LegUp* this is not a real problem because of their open source licenses. This allows to modify the memory allocation pass of these compilers to obtain the data. A preliminary feasibility study on both these open source tools showed that this was possible for both, with the same fundamental approach. We decided to implement the full fledged version of the debugger only for *bambu*, because it has a very complex memory model, allowing tests in more challenging cases. For this reason the results on coverage and bug finding in the following sections are strictly related to *bambu*. For *LegUp*, we applied the algorithm manually on the CHStone benchmarks, to check that the methodology was actually portable even if we did not deploy the full-fledged automated flow.

For *CTool* some additional work was necessary to build the ASTS and the `decodeHW` function, because the HLS flow could not be altered. We compiled some of the CHStone benchmarks that the tool was able to synthesize, using memory partitioning directives. We had to chose a design for which *CTool* generate correct address decoding, because if we were to infer the ASTS and the `decodeHW` function from the HDL, we had to be sure that they were correct before injecting

the bug. For this reason, we had to restrict the analysis of *CTool* to examples where the generated designs were correct and passing all the functional tests. This allowed manual analysis of the generated HDL to build the correct ASTS and the `decodeHW` function. After building them, the HDL of the address decoding logic generated by *CTool* was altered manually to introduce bugs. This operation showed that it is possible to use the correct ASTS and `decodeHW`, built in advance, to spot a hypothetical bug inserted by the tool, manually applying our algorithm to the fault-injected design. Obviously, the problems analyzed in this way are a subset of all the possible cases. Nevertheless, the results are encouraging because they show that the method is successfully applicable even to commercial HLS flows. In our experiments with *CTool* we had to resort to manual methods, but with the access to the source code it should be easy to adapt the tool to implement an automated debug flow as described for *bambu*. This has been valuable to show that the methodology can target commercial HLS tools and that it can handle memory options not available in *bambu*, like array partitioning.

C. Detected Bugs

Applying extensively the described approach to *bambu* we were able find several bugs involving pointer operations. Interestingly, the bugs detected with this method are not always strictly caused by errors in memory allocation. Instead they can be generated by problems in other steps of HLS. The one thing they have in common is that they affect in some way the address decoding logic of the generated circuit, causing a wrong address to be computed at some point. Here is an exemplifying but not exhaustive list of the affected steps in HLS, with some of the found bugs.

1) Compiler Frontend: bugs due to wrong static analysis or manipulations of the IR, before the actual HLS takes place. Among them, a compiler pass in *bambu* performs bit-width static analysis to reduce the bits necessary for addressing memories. This step was buggy and sometimes the number of bits that the tool required to be necessary to represent addresses was too high or too low. In both cases the effect was that wrong values were used to address the memory, causing bugs that propagated to the rest of the design.

2) Scheduling: problems due to wrong scheduling of operations in the FSM. They include wrong reordering of operations due to missing dependencies, bad scheduling due to wrong computations of operations' execution times and others. In some cases *bambu*'s frontend lost information about data dependencies among operations. As a consequence, the scheduling step in HLS decided that an address could be computed in advance, but the data used for the computation was actually not ready to use, again generating wrong addresses.

3) Memory allocation: instantiation of memory modules with wrong characteristics, size or latency. This happened with *bambu* and it caused different kind of problems. When the memory was too small, some data could be lost writing it to an out-of-bound address. It also happened that the HW tried to read data from an out-of-bound address, causing the design to hang, waiting for a reply from memory that never happened. When the memory was too large, the offset calculation in address decoding was wrong. This caused memory accesses at wrong locations, reading wrong data or writing them in the wrong place. Finally, when the expected latency was wrong, the HW was using data before the memory replied.

4) Interconnection: wrong connection of wirings, causing malfunctions. For instance, the same bug described at point (1) affected the generation of the interconnection. Thus, the address bus had the wrong width, causing wrong addressing.

All these bugs were properly detected and isolated in *bambu* by the approach, without human intervention. The data provided by the discrepancy analysis engine allowed to identify the cause and to fix the HLS compiler. Positively, the approach was able to treat bugs causing the designs to hang or loop forever, because the simulation could be interrupted to perform the discrepancy analysis up to a certain point in the execution. Another important thing to stress is that most of the memory bugs detected with the discrepancy analysis were also causing errors in variables that did not represent addresses. Clearly if a READ loads data from the wrong location, it is likely to get them wrong. Without the address discrepancy analysis it would not be possible to know if the problem was the address or the data in the memory itself. The GCC C-torture tests were very valuable in this phase, because the CHStone benchmarks did not trigger any of these bugs.

D. Coverage and Performance

The results shown in Table I have been collected with *bambu* on both CHStone and GCC C-torture tests. For the CHStone the benchmark names are reported with the used optimization flags. For GCC C-torture we report aggregated data, because the benchmark set is too big to fit entirely here. The results are measured with respect to the *instruction coverage* icov described in section II-C. For the sake of completeness, in the first two columns of Table I we report the dynamic *statement coverage* for `C` ('ccov') and Verilog ('vcov'). For `C`, ccov was computed with `gcov`, the coverage tool included in GCC-4.9. For Verilog, vcov was evaluated with simulation, using ModelSim. The data show that the tests are not covering the whole design, nor the original program. This is expected, since the data set on which the CHStone operate is fixed and it was not designed with coverage in mind. Currently, the discrepancy analysis does not consider how the inputs for the design under test are generated. For the experiments the default was used. Despite this, the address discrepancy analysis was able to find several bugs, because of its per-operation granularity. However, the problem of how to tune the input test is interesting, especially because there is no straightforward relationship between coverage in `C` and in Verilog. The topic is orthogonal to what is described here. It is very vast and it deserves a separate analysis.

The interesting columns are 'icov' and 'res'. Column 'icov' reports the static *instruction coverage*. For the CHStone it is always higher than 70%, with a peak of 97%. As explained in section II-C, this includes only data operations that are

benchmark	ccov	vcov	icov	res	addr	perf
adpcm-O0	100%	73.7%	70.7%	78.4%	22.5%	10.3%
adpcm-O3	99.68%	72.6%	86.6%	100%	5.6%	5.1%
aes-O0	69.53%	62.5%	77.6%	89.7%	42.0%	1.9%
aes-O3	72.46%	64.0%	81.2%	99.3%	20.8%	0.7%
bf-O0	67.95%	61.4%	95.2%	61.7%	26.4%	189.5%
bf-O3	100%	75.9%	92.0%	65.2%	20.1%	38.7%
dfadd-O0	78.44%	76.7%	74.8%	15.0%	1.8%	0.7%
dfadd-O3	88.39%	64.7%	96.9%	100%	0.9%	0.7%
dfmul-O0	83.27%	71.9%	90.5%	100%	1.4%	0.1%
dfmul-O3	78.84%	65.3%	96.3%	100%	1.1%	0.5%
dfdiv-O0	76.00%	66.1%	90.0%	100%	2.1%	0.1%
dfdiv-O3	86.67%	69.0%	97.2%	100%	1.3%	0.1%
dfsin-O0	74.36%	53.1%	78.3%	10.5%	0.4%	6.3%
dfsin-O3	80.12%	48.0%	89.2%	100%	0.2%	27.0%
gsm-O0	96.41%	73.8%	87.2%	94.0%	22.3%	4.2%
gsm-O3	91.07%	70.1%	89.1%	95.0%	25.5%	1.9%
jpeg-O0	89.40%	89.4%	77.4%	91.2%	19.9%	87.6%
jpeg-O3	85.40%	88.0%	83.6%	88.9%	22.4%	124.0%
mips-O0	74.85%	51.9%	75.6%	100%	0.3%	3.5%
mips-O3	73.41%	50.0%	76.2%	100%	0.2%	4.7%
mpeg2-O0	88.67%	79.6%	80.5%	54.5%	32.8%	1.1%
mpeg2-O3	85.17%	78.6%	78.8%	70.0%	28.3%	0.4%
sha-O0	98.81%	75.3%	81.4%	100%	43.8%	92.6%
sha-O3	87.15%	64.5%	77.3%	93.6%	33.6%	71.3%
C-torture	80.89%	74.7%	48.7%	28.5%	28.8%	0%

TABLE I: ccov and vcov: statement coverage for C and Verilog; icov: instruction coverage; res: percentage of pointers for which alias-analysis is fully resolved; addr: percentage of address operations; perf: execution time overhead compared to simulation

directly checked by the discrepancy analysis, even if control flow operations are checked indirectly by the methodology. Column 'res' reports the percentage of pointers in the C code for which the alias-analysis is fully resolved at compile time, for which the discrepancy analysis is more effective. If a bug affects one of them, the Algorithm 1 is always able to find it by design. When the analysis is not fully resolved, the address discrepancy analysis may have to give up in some cases. If the analyzed pointer is not in range for any memory-allocated variable (a so called wild pointer), Algorithm 1 never returns a mismatch (lines 14-16). The reason is that only in-range memory locations are mapped in HW, thus out-of-bound addresses cannot be checked. ASAN is used to ensure that there are no wild pointer dereferences, but it is interesting to measure how many times the address discrepancy analysis actually gives up on a comparison. From our inspection, this never happens for the CHStone tests. This means that even if the alias analysis is non-trivial the ASTS is able to check all the addresses. Interestingly the percentage of give up is very small (0.004%) also for the GCC C-torture, even if they are precisely designed to stress-test the compiler on pointer arithmetic. The good results on the CHStone benchmarks show that the approach is perfectly suitable for real use cases. On the other hand, the fact that even on the GCC C-torture test there are so few give ups confirms that the ASTS can handle a large variety of situations including several corner cases of allowed operations with C pointers.

In column 'addr' we report the percentage of the operations that assign addresses on the total operations executed at runtime. For memory-intensive applications it can be higher than 40%, so the address discrepancy is a non-trivial problem, especially in large designs. Without the ASTS, the developers would need to reconstruct the address translation map manually. This takes a long time and it requires memory allocation information from the HLS tool, with details of the addresses, the memory alignment and what are the bit patterns used as addresses in HW. Also the SW memory map with stack-allocated data should be retrieved. Finally, the Algorithm 1 would have to be executed manually. In particular, the user would need to decode manually the values of the HW signals to retrieve the memory locations, which is an operation that requires a good understanding of the internal memory allocation used in HLS. All these operations could be automated with the proposed approach.

The performance of the address discrepancy analysis is measured in terms of execution time overhead compared to simulation, that was executed with ModelSim SE-64 10.3. This overhead is reported in the column 'perf'. When the number of address operation is negligible (dfadd, dfmul, dfdiv, dfsin, mips) the execution time overhead is not significant This is reasonable, because the discrepancy analysis is very fast on integers and floating points, since there are no ASTS lookups. For the GCC C-torture tests the overhead is also negligible, because they are really small programs. On the other hand, when the address operations are more than about 20%, it is harder to find a straightforward relationship between the coverage metrics and the performance overhead. The main reason is to be attributed to the results of the alias analysis. Indeed, even if the alias analysis is fully resolved it may be possible that a single pointer variable points to different memory location (if the location set for that pointer has more than one element). Things are even worse if the alias-analysis is not resolved. These situations can have a big impact on the execution time of Algorithm 1, because several lookups in the ASTS may be required. Even in the worst cases the overhead is less than 200%, which is not a huge overhead if we consider the complexity of the address discrepancy analysis and the time we save avoiding user interaction.

VI. Conclusions and Future Work

The focus of this work was to provide a methodology for debugging circuits generated by HLS, able to handle pointers and address comparisons and not tied to a specific architecture or memory layout of the generated HW. The described approach is able to isolate a bug at the operation level, showing good coverage and a percentage of give up close to zero even in corner cases. The described approach has proved to be accurate and general enough to be adopted by commercial HLS tools, while not imposing unrealistic bounds to compiler optimizations for memory allocation. Given the reduced interaction with the user, it is also an interesting option for the development of environments relying on continuous integration and regression testing to readily spot bugs. Further research could try to extend the methodology to support debugging of multi-threaded programs synthesized in HW or READ speculation.

REFERENCES

[1] Y. Ben-Asher and N. Rotem, "Using memory profile analysis for automatic synthesis of pointers code," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 3, p. 68, 2013. [Online]. Available: http://doi.acm.org/10.1145/2442116.2442118

[2] C. Pilato, F. Ferrandi, and D. Sciuto, "A design methodology to implement memory accesses in high-level synthesis," in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, 2011, pp. 49–58. [Online]. Available: http://doi.acm.org/10.1145/2039370.2039381

[3] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, 2014, pp. 199–208. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554780

[4] P. Zhang, M. Huang, B. Xiao, H. Huang, and J. Cong, "CMOST: a system-level FPGA compilation framework," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, 2015, pp. 158:1–158:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744807

[5] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, 2013, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488748

[6] J. Seo, T. Kim, and P. R. Panda, "Memory allocation and mapping in high-level synthesis - an integrated approach," *IEEE Trans. VLSI Syst.*, vol. 11, no. 5, pp. 928–938, 2003. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2003.817116

[7] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "System-level memory optimization for high-level synthesis of component-based socs," in *2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12-17, 2014*, 2014, pp. 18:1–18:10. [Online]. Available: http://doi.acm.org/10.1145/2593069.2500071

[8] L. Séméria and G. D. Micheli, "Resolution, optimization, and encoding of pointer variables for thebehavioral synthesis from C," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 213–233, 2001. [Online]. Available: http://dx.doi.org/10.1109/43.908442

[9] N. Calagar, S. D. Brown, and J. H. Anderson, "Source-level debugging for FPGA high-level synthesis," in *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, 2014, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/FPL.2014.6927496

[10] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. S. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, p. 24, 2013. [Online]. Available: http://doi.acm.org/10.1145/2514740

[11] P. Fezzardi, M. Castellana, and F. Ferrandi, "Trace-based automated logical debugging for high-level synthesis generated circuits," in *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, 2015, pp. 251–258. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2015.7357111

[12] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO) 2004, 20-24 March 2004, San Jose, CA, USA*, 2004, pp. 75–88. [Online]. Available: http://dx.doi.org/10.1109/CGO.2004.1281665

[13] GNU GCC compiler. `https://gcc.gnu.org`.

[14] M. Lattuada and F. Ferrandi, "Code transformations based on speculative SDC scheduling," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*, 2015, pp. 71–77. [Online]. Available: http://dx.doi.org/10.1109/ICCAD.2015.7372552

[15] B. Steensgaard, "Points-to analysis by type inference of programs with structures and unions," in *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, 1996, pp. 136–150. [Online]. Available: http://dx.doi.org/10.1007/3-540-61053-7_58

[16] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, 1995, p. 1. [Online]. Available: http://doi.acm.org/10.1145/207110.207111

[17] A. Ribon, B. L. Gal, C. Jégo, and D. Dallet, "Assertion support in high-level synthesis design flow," in *2011 Forum on Specification & Design Languages, FDL 2011, Oldenburg, Germany, September 13-15, 2011*, 2011, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6069472

[18] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis," *Int. J. Reconfig. Comp.*, vol. 2011, pp. 406 857:1–406 857:17, 2011. [Online]. Available: http://dx.doi.org/10.1155/2011/406857

[19] M. B. Hammouda, P. Coussy, and L. Lagadec, "A design approach to automatically synthesize ANSI-C assertions during high-level synthesis of hardware accelerators," in *IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, June 1-5, 2014*, 2014, pp. 165–168. [Online]. Available: http://dx.doi.org/10.1109/ISCAS.2014.6865091

[20] J. B. Goeders and S. J. E. Wilton, "Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs," in *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, 2015, pp. 127–134. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2015.25

[21] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005*, 2006.

[22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, 2012, pp. 309–318.

[23] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *International Symposium on Circuits and Systems (ISCAS) 2008, 18-21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA*, 2008, pp. 1192–1195. [Online]. Available: http://dx.doi.org/10.1109/ISCAS.2008.4541637

[24] GCC C-torture tests. `https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.