Precise Pointer Analysis in High-Level Synthesis

Nadesh Ramanathan, George A. Constantinides and John Wickerson
Department of Electrical and Electronic Engineering
Imperial College London, London, SW7 2AZ, United Kingdom
E-mail: {n.ramanathan14, g.constantinides, j.wickerson}@imperial.ac.uk

Abstract—Pointer analysis computes the set of memory locations that each pointer access can point to during hardware runtime. The more precise the pointer analysis, the more precise these sets are likely to be, reducing unnecessary sharing of memory resources between instructions during high-level synthesis (HLS) memory generation. Despite the importance of precision, modern HLS tools typically sacrifice precision to prioritise quicker analysis times, although there are several pointer analyses that can produce reasonably precise points-to sets within an acceptable amount of time. In this paper, we explore the effects of precise pointer analysis within a modern HLS tool (LegUp) on a set of benchmark programs (PTABen) that are challenging to its original pointer analysis. We see precise analysis that reduces unnecessary memory sharing, leading to average LUT savings of 60% and runtime improvements of 42%.

I. INTRODUCTION

Pointer analysis determines the set of memory locations that each pointer-related instruction can point to, referred to as its points-to set. When high-level synthesis (HLS) tools synthesise programs with pointers, these points-to sets influence memory synthesis in terms of sharing of memory resources. The more precise these points-to sets, the more likely the HLS tool generates simple addressing circuitry between instructions and memory resources.

The precision of pointer analysis can be improved by making the analysis sensitive to certain features of the program. The two common sensitivities [1], [2] that pointer analysis can consider are flow and context [3]–[17]. Flow-sensitivity considers the order in which memory operations are executed whereas context-sensitivity considers the calling context of functions. Although fully precise pointer analysis is undecidable [18], various precise pointer analyses can refine points-to precision within an acceptable time on large codebases.

A common pointer analysis adopted by modern HLS tools is Andersen analysis [19], which is a flow-insensitive context-insensitive analysis. The output of Andersen analysis is instruction-agnostic, i.e. Andersen analysis only relates variables. Hence, Andersen analysis can lead to over-approximation of points-to relation between instructions and variables, inducing unnecessary memory sharing within HLS.

Imprecise pointer analyses are especially attractive to modern HLS tools, since these tools tend to overlook precision in favour of faster analysis times. For example, LegUp HLS uses a variant of Andersen analysis [20], as it claims that the compiler community has developed fast insensitive analyses [21] §4.11. Bambu HLS [22] also uses this variant of Andersen analysis. Vivado HLS [23] restricts non-trivial use of pointers and typically converts pointer instructions into static LLVM loads or stores.

In practice, the overheads of precise pointer analysis are less problematic for HLS compilers because input programs tend to be smaller, and synthesis times are much longer than pointer analysis times. Séméria et al. [24] and Zhu et al. [13] claim to support precise pointer analysis within HLS but they do not evaluate the impact of precision on hardware quality or analysis times, both of which we address. Recent HLS works on synthesising pointer-manipulating programs [25], [26], atomic pointers [27], [28] and dynamic memory allocation [29]–[32] are examples of non-trivial use of pointers, which will increase the need for emphasis on points-to precision in the future.

In this paper, we leverage an existing flow-sensitive context-sensitive pointer analysis tool within a modern HLS compiler. In §II, we provide an example in which precise analysis improves the quality of HLS-generated hardware. In §III we augment LegUp HLS [21] to utilise the flow- and context-sensitive SVF pointer analysis [15], [16], instead of Andersen analysis. In §IV we demonstrate that SVF’s precise analysis improves the quality of hardware generated by LegUp on a set of programs from the PTABen benchmark suite [33], which uses pointers non-trivially. When precise pointer analysis is applied, the hardware generated for these programs has an average LUT saving of 60% and runtime improvement of 42%. We also show that, although improving precision incurs analysis time overheads, these overheads are mostly negligible since the analysis times are within tens of milliseconds.

II. MOTIVATING EXAMPLE

In this section, we discuss an example that shows how precise analysis influences the points-to relation between instructions and memory variables in HLS. Consider the program in Fig. 1a which consists of four statements with a pointer, p, and two variables, a and b. In the main function, the first statement stores the address of a to pointer p. The second statement calls function f, within which the third statement stores the address of b to pointer p. Finally, the fourth statement dereferences pointer p and returns its value. We disable function inlining to avoid optimisations.

An LLVM-based HLS tool, such as LegUp, compiles this C program into LLVM IR code similar to that shown in Fig. 1b. The LLVM IR code contains the LLVM loads #1 and #2 directly address p. The dereferencing of *p in the C program is compiled to two LLVM loads [34]. The first load #1 directly addresses p and the second load #2 indirectly addresses the value loaded from...
int *p, a = 10, b = 20;  
__attribute__((noinline))
void f() {  p = &b; }

int main() {  
p = &a;
f();  
return (*p);
}

(a) a program

(b) compiled LLVM IR

(c) imprecise analysis

(d) precise analysis

(e) sub-optimal hardware

(f) optimal hardware

Fig. 1. An example program to LLVM IR, which is provided to insensitive Andersen analysis and precise analysis, whose results are utilised by LegUp to generate hardward. In Figs. 1c and 1d black and red edges are points-to relation for directly and indirectly accessed memory instructions. Blue arrows in Fig. 1 are the results on Andersen analysis. Figs. 1e and 1f show the memory architecture that is generated by LegUp based on the different pointer analyses.

In this section, we describe how we augment the LegUp HLS tool [21] to utilise SVF’s flow-sensitive context-sensitive pointer analysis. LegUp HLS is built on the LLVM framework and converts a C program to Verilog, via a series of HLS transformations followed by a Verilog backend generator. Originally, LegUp applies Andersen analysis [19] whose results it uses to generate memory addressing between instructions and variables and also allocates all variables into different LegUp memories. We discuss how LegUp performs both these tasks, in §III-A. Then, in §III-B we discuss how the output of SVF’s precise pointer analysis can be utilised by LegUp to perform the same tasks and how we implement SVF in LegUp.

A. Understanding LegUp’s insensitive pointer analysis

Let V be the set of variables in the IR code. Andersen analysis produces a relation between variables, $AnderPts \subseteq V \times V$. For example, $AnderPts = \{(p, a), (p, b)\}$ based on the IR code in Fig. 1b as shown by the blue arrows in Fig. 1c.

1) Memory addressing: LegUp uses the results of Andersen analysis to generate memory addressing for all LLVM memory instructions. Let I be the set of LLVM memory instructions. LegUp defines a points-to relation $InstPts$ between instructions and variables, i.e. $InstPts \subseteq I \times V$, as follows:

$$InstPts = \{ (i, v) \mid i \in I \land v \in V \land \exists \varepsilon : (\varepsilon, v) \in AnderPts \}.$$
DirectPts is a relation that represents instructions that directly address variables, where this relation can be obtained from the LLVM source. For example, DirectPts = \{(i,p),(n,p),(o,p)\} can be obtained from the IR code in Fig. 1b shown as black edges in Fig. 1c. IndirectPts is a relation that represents instructions that indirectly address variables, where this relation is inferred from Andersen analysis. IndirectPts defines that an instruction i points to variable v, if instruction i dereferences a pointer vp, i.e. (i,vp) ∈ deref, and Andersen analysis states that vp points to v. For example, for the code in Fig. 1b, SVF generates InstPts = FSIInstPts = \{(n,p),(o,p),(e,p),(e,b)\} since SVF understands that latest address value written to p before it is dereferences is b, as shown in Fig. 1d. Due to this refinement of InstPts, LegUp can infers that LocalMem = \{p,b\} and GlobalMem = ∅. Hence, the addressable memory controller can be removed, all variables can be connected directly to instructions that point to them and a can be removed since it is never pointed to by any instruction, as seen in Fig. 1f.

2) Memory allocation: Subsequently, LegUp also uses InstPts to allocate memory. LegUp can generate two types of memories: local and global. Local memories (LocalMem) consists of variables that are connected directly to instructions, whereas global memories (GlobalMem) consists of variables that are accessed by instructions via an addressable memory controller. LegUp defines local and global memories is defined as follows:

GlobalMem = \{v \mid v \in V \land \exists i \in I, \exists v' \in V, v \neq v' \land (i,v) \in InstPts \land (i,v') \in InstPts\}.

LocalMem = V \setminus GlobalMem

where any variable v is implemented in global memory (GlobalMem) if an instruction i points to not only v but at least one other variable v'. If all memory instructions that point to v do not point to any other variable, then v is implemented in local memory that is directly accessible without the memory controller. For example, instPts = \{(n,p),(o,p),(e,p),(e,a),(e,b)\} from code in Fig. 1b as shown in Fig. 1c. Based on InstPts, LegUp infers that LocalMem = \{p\} and GlobalMem = \{a,b\}, since e, o and p only point to p whereas c can point to both a and b, as shown in Fig. 1f.

B. Leveraging SVF’s precise pointer analysis within LegUp

SVF’s precise analysis produces a points-to relation between all LLVM memory instructions and variables, i.e. I × V. We can configure SVF either as a flow-sensitive analysis (FSInstPts ⊆ I × V) or as a flow- and context-sensitive analysis (FSCSInstPts ⊆ I × V). Additionally, FSInstPts ⊆ FSInstPts ⊆ FSInstPts since SVF’s flow-sensitive analysis takes Andersen analysis as input and SVF’s flow- and context-sensitive analysis takes its flow-sensitive analysis as input.

1) LegUp’s interpretation of SVF output: LegUp can directly utilise the points-to relation of SVF for memory addressing and allocation, i.e. InstPts = FSInstPts or InstPts = FSCSInstPts. The difference between using SVF’s precise analysis and Andersen analysis in HLS is that SVF directly provides HLS with the points-to results in the form of I × V. However, when using Andersen analysis, the HLS tool needs to explicitly translate the output of Andersen analysis (V × V) into a usable points-to results for HLS (I × V). This translation is the main cause for over-approximation of points-to precision within HLS, leading to unnecessary memory sharing.

For example, for the code in Fig. 1b, SVF generates InstPts = FSInstallPts = \{(n,p),(o,p),(e,p),(e,b)\} since SVF understands that latest address value written to p before it is dereferences is b, as shown in Fig. 1d. Due to this refinement of InstPts, LegUp can infer that LocalMem = \{p,b\} and GlobalMem = ∅. Hence, the addressable memory controller can be removed, all variables can be connected directly to instructions that point to them and a can be removed since it is never pointed to by any instruction, as seen in Fig. 1f.

IV. Evaluation

In this section, we evaluate precise analysis on a set of programs with non-trivial and challenging use of pointers, from the PTABen benchmark suite [33]. Our evaluation focuses on two key questions, which are 1) To what extent does precise pointer analysis affect the quality of hardware generated by LegUp HLS? 2) What, if any, is the added cost in terms of analysis times to adopt precise pointer analysis within HLS?

a) Experimental setup: We evaluate all programs on three design points. The first design point, IA, is LegUp’s original insensitive Andersen analysis. The second design point, FS, is a flow-sensitive SVF analysis implemented within LegUp. The third design point, FSACS, is a flow- and context-sensitive SVF analysis implemented within LegUp. We utilise LegUp’s pure hardware, which allocates C memories as FPGA registers or RAMs. Our synthesis tool is Quartus v15.0, which targets a Cyclone V FPGA.

b) Selecting and modifying programs from PTABen:

The PTABen benchmark suite comprises over 400 hand-written programs that tests for correctness and precision of pointer analyses, all of which are relatively new to the HLS community. We identified 50 programs whose objective is to test the flow- and context sensitivity of pointer analysis (two subfolders). Out of these 50 programs, we are able to synthesise 32 programs since they do not require dynamic memory allocation or recursion. We minimally modify these 32 programs from PTABen for our purposes. We replace PTABen’s backdoor calls to check the precision of points-to sets of various pointer instructions with non-inlined functions that dereference these pointers, which enables hardware instrumentation of points-to set within HLS-generated hardware.

1) Results of synthesising PTABen programs: Fig. 2 shows the effects of precise pointer analysis on the set of programs we synthesise from the PTABen benchmark suite.

a) Points-to ratio: Fig. 2a shows the points-to ratio of the different pointer analyses, which is the number of points-to relation, [InstPts], divided by the number of instructions, |I|. The best achievable ratio is one, whereby every memory instruction points to one location. We see that the IA’s points-to ratio is always higher than or equal to the points-to ratios of FS or FSACS. On average, FS and FSACS analyses reduces points-to ratio by 11% and 17% respectively, compared to IA.
### b) Hardware resources

Typically, the reductions in points-to ratio results in hardware with smaller area. Fig. 2b shows the LUT savings of FS and FSCS relative to the LUT utilisation of IA. FSCS reduces the LUT utilisation of 25 out of 32 programs, where these reductions come from either removing the memory controller and avoiding unnecessary memory sharing. On average, FS and FSCS analyses reduce LUT utilisation by 28% and 60%, with maximum of 86% and 97%, respectively compared to IA.

Although precise analysis reduces the points-to ratio of 29 programs, they are four cases where LegUp’s hardware generation does not take advantage of this refinement. This is typically because LegUp generates the same GlobalMem for all three analyses, despite FSCSInstPts ⊆ FSInstPts ⊂ InstPts. This discrepancy between points-to ratio reduction and resource reduction suggests that current HLS memory generation may be supra-optimal for non-trivial pointer use.

### c) Hardware runtimes

In addition to LUT savings, precise pointer analysis also improve hardware runtimes, since avoiding the memory controller and unnecessary memory sharing improve access latencies and clock frequencies. Fig. 2c show the hardware speedups gained by FS and FSCS relative to hardware runtimes of IA. On average, FS and FSCS analyses improve hardware runtimes by 17% and 42%, with maximum of $2\times$ and $2.6\times$, respectively compared to IA.

### d) Analysis time overheads

Fig. 2d shows the analysis times of all three analyses, where IA analysis is always the fastest. The wall-clock times of all these analyses are in the range of milliseconds, which suggests that the cost of employing more precise pointer analysis is insignificant compared to hardware synthesis.

## V. Conclusion

In this paper, we evaluate the effects of precise pointer analysis within the context of HLS. We augment the LegUp HLS tool to utilise a flow- and context-sensitive pointer analysis. Then, we evaluate both the effects of insensitive and precise analyses on programs from the PTABen benchmark suite. Our evaluation demonstrates that there exist programs where sensitive pointer analysis can lead to significantly improved hardware, at the cost of a few extra milliseconds of compilation time. On average, precise pointer analysis reduces LUT utilisation by 60%, with a maximum of up to 97%. Overall, we show that, for programs with non-trivial use of pointers, precision of pointer analysis plays an important role in reducing unnecessary memory sharing. As the complexity of pointer-based programs that are synthesisable via HLS increases, points-to precision will be increasingly important. We hope that this work acts as the catalyst to explore future directions in synthesising pointer-based programs.

### Acknowledgements

We would like to thank Yulei Sui for providing timely insights on the SVF tool. This work is supported by EPSRC (EP/P010040/1).