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 sensitive 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 stores ❶ and ❷ directly address `p`. The dereferencing of `*p` in the C program is compiled to two LLVM loads [34]. The first load ❸ directly addresses `p` and the second load ❹ 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);
}
```

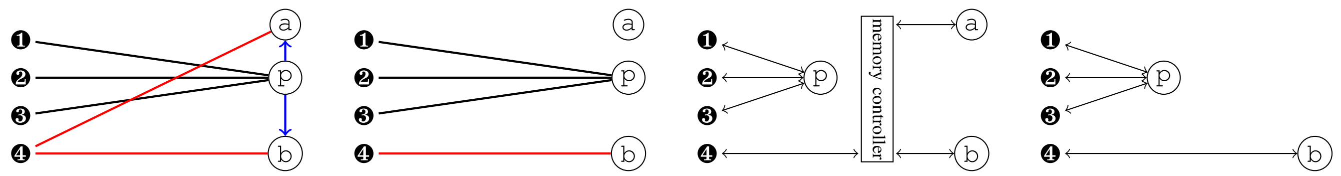
(a) a program

```
@a = global i32 10
@b = global i32 20
@p = global i32* null

void @f(){
  store i32* @b, i32** @p ❷
  ret void
}
```

```
i32 main() {
  store i32* @a, i32** @p ❶
  call void @f()
  %1 = load i32** @p ❸
  %2 = load i32* %1 ❹
  ret i32 %2
}
```

(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 hardware. In Figs. 1c and 1d black and red edges are points-to relation for directly and indirectly accessed memory instructions. Blue arrows in Fig. 1c 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.

p, via the label %1 of ❸. Since ❹ indirectly addresses memory variables via p, pointer analysis is required to determine all variables that ❹ can point to.

a) Insensitive Andersen analysis: Andersen analysis infers that any dereferencing of p can point to either a or b during runtime. Andersen analysis relates variables without considering any instructions, i.e. $\{(\mathtt{p}, \mathtt{a}), (\mathtt{p}, \mathtt{b})\}$, as shown by the blue arrows in Fig. 1c. Based on this output, the HLS tool infers ❹ may point to either a or b, i.e. $\{(❹, \mathtt{a}), (❹, \mathtt{b})\}$, as shown by the red edges in Fig. 1c. Since, ❹ can point to two possible locations, both these memory elements must be connected in an addressable manner. LegUp supports this possibility via its memory controller, as shown in Fig. 1e.

b) Flow-sensitive pointer analysis: A flow-sensitive analysis understands the order in which the accesses to p happen: ❶ < ❷ < ❸. Hence, when ❸ occurs, the pointer analysis is aware that the value of p has been updated to the address of b. Hence, such an analysis can conclude that ❹ can only access b during hardware runtime, i.e. $\{(❹, \mathtt{b})\}$, as shown by the red edges in Fig. 1d. This refined points-to relation can result in simpler memory circuitry for the same program. For example, when LegUp can identify that ❹ can only point to exact one location, b, it directly connects ❹ to b, as shown in Fig. 1f. Additionally, LegUp can also infers that a can be removed since it is never accessed.

c) Context-sensitive pointer analysis: Suppose we replace the function f with `int* f(int *q){ return q;}` and its callee with `p = f(&b);`. For this modified example, a flow-sensitive analysis generates an imprecise points-to set for ❹, as in Fig. 1c, because it cannot understand the context in which function f is called. When a context-sensitive analysis is applied to this modified example, a precise points-to set of ❹ can be obtained, as in Fig. 1d.

III. Method

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, $\mathit{AnderPts} \subseteq V \times V$. For example, $\mathit{AnderPts} = \{(\mathtt{p}, \mathtt{a}), (\mathtt{p}, \mathtt{b})\}$ 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. $\mathit{InstPts} \subseteq I \times V$, as follows:

$$\mathit{InstPts} = \mathit{DirectPts} \cup \mathit{IndirectPts}$$

where

$$\mathit{IndirectPts} = \{(i, v) \mid i \in I \wedge v \in V \wedge \exists v_p \in V. \\ (i, v_p) \in \mathit{deref} \wedge (v_p, v) \in \mathit{AnderPts}\}.$$

$\mathit{DirectPts}$ is a relation that represents instructions that directly address variables, where this relation can be obtained from the LLVM source. For example, $\mathit{DirectPts} = \{(❶, \mathtt{p}), ❷, \mathtt{p}), ❸, \mathtt{p})\}$ can be obtained from the IR code in Fig. 1b, shown as black edges in Fig. 1c. $\mathit{IndirectPts}$ is a relation that represents instructions that indirectly address variables, where this relation is inferred from Andersen analysis. $\mathit{IndirectPts}$ defines that an instruction i points to variable v, if instruction i dereferences of a pointer v_p, i.e. $(i, v_p) \in \mathit{deref}$, and Andersen analysis states that v_p points to v. For example, since $\mathit{AnderPts} = \{(\mathtt{p}, \mathtt{a}), (\mathtt{p}, \mathtt{b})\}$ and $(❹, \mathtt{p}) \in \mathit{deref}$, ❹ must be related to both $\mathtt{a}$ and $\mathtt{b}$ i.e. $\mathit{IndirectPts} = \{(❹, \mathtt{a}), (❹, \mathtt{b})\}$. $\mathit{DirectPts}$, shown as red edges in Fig. 1c. Together, $\mathit{DirectPts}$ and $\mathit{IndirectPts}$ form a points-to relation that LegUp uses for memory addressing.

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

$$\begin{aligned}\mathit{GlobalMem} = \{v \mid\ & v \in V \wedge \exists i \in I \,.\, \exists v' \in V \,.\, v \neq v' \wedge \\ & (i, v) \in \mathit{InstPts} \wedge (i, v') \in \mathit{InstPts}\}. \\ \mathit{LocalMem} =\ & V \setminus \mathit{GlobalMem}\end{aligned}$$

where any variable v is implemented in global memory ($\mathit{GlobalMem}$) 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, $\mathit{instPts} = \{(❶, \mathtt{p}), (❷, \mathtt{p}), (❸, \mathtt{p}), (❹, \mathtt{a}), (❹, \mathtt{b})\}$ from code in Fig. 1b, as shown in Fig. 1c. Based on $\mathit{InstPts}$, LegUp infers that $\mathit{LocalMem} = \{\mathtt{p}\}$ and $\mathit{GlobalMem} = \{\mathtt{a}, \mathtt{b}\}$, since ❶, ❷ and ❸ only point to $\mathtt{p}$ whereas ❹ can point to both $\mathtt{a}$ and $\mathtt{b}$, as shown in Fig. 1e.

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. $I \times V$. We can configure SVF either as a flow-sensitive analysis ($\mathit{FSInstPts} \subseteq I \times V$) or as a flow- and context-sensitive analysis ($\mathit{FSCSInstPts} \subseteq I \times V$). Additionally, $\mathit{FSCSInstPts} \subseteq \mathit{FSInstPts} \subseteq \mathit{InstPts}$, 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. $\mathit{InstPts} = \mathit{FSInstPts}$ or $\mathit{InstPts} = \mathit{FSCSInstPts}$. 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 $I \times V$. However, when using Andersen analysis, the HLS tool needs to explicitly translate the output of Andersen analysis ($V \times V$) into a usable points-to results for HLS ($I \times V$). 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 $\mathit{InstPts} = \mathit{FSInstPts} = \{(❶, \mathtt{p}), (❷, \mathtt{p}), (❸, \mathtt{p}), (❹, \mathtt{b})\}$ since SVF understands that latest address value written to $\mathtt{p}$ before it is dereferences is $\mathtt{b}$, as shown in Fig. 1d. Due to this refinement of $\mathit{InstPts}$, LegUp can infers that $\mathit{LocalMem} = \{\mathtt{p}, \mathtt{b}\}$ and $\mathit{GlobalMem} = \emptyset$. Hence, the addressable memory controller can be removed, all variables can be connected directly to instructions that point to them and $\mathtt{a}$ 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, *FSCS*, 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 handwritten 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, $|\mathit{InstPts}|$, 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 *FSCS*. On average, *FS* and *FSCS* analyses reduces points-to ratio by 11% and 17% respectively, compared to *IA*.

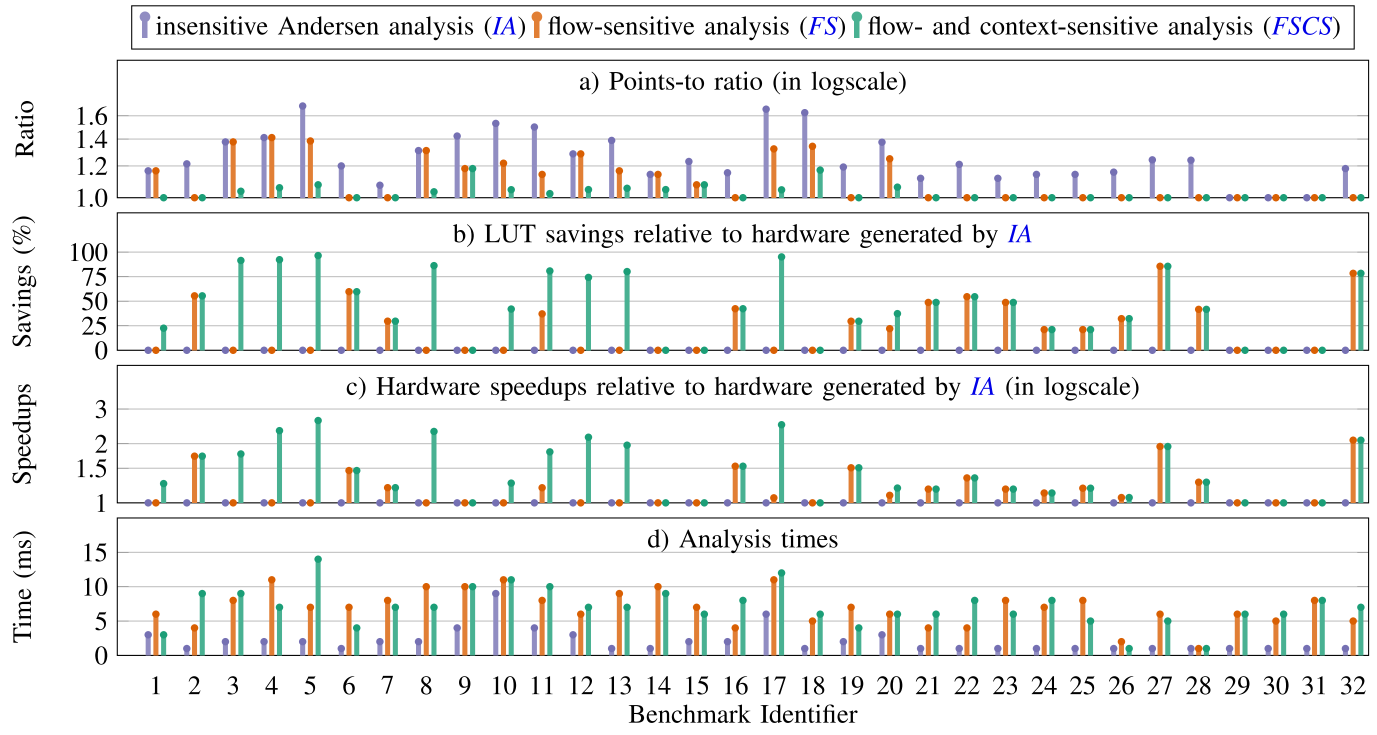


Fig. 2. Pointer analysis metrics by programs and by pointer analysis type. We enumerate the results of synthesising the following PTABen programs (from `cs_tests` and `fs_tests`) in this order: `cs0-15.c`, `cs17-20.c`, `simple1-3.c`, `global1-5.c`, `branch1-3.c` and `strong-update.c`.

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 $\mathit{FSCSInstPts} \subseteq \mathit{FSInstPts} \subset \mathit{InstPts}$. This discrepancy between points-to ratio reduction and resource reduction suggests that current HLS memory generation may be sup-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).

References

[1] M. Hind and A. Pioli, "Which pointer analysis should I use?" in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000, pp. 113–123.

[2] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[3] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural aliasing," *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 235–248, 1992.

[4] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp. 232–245.

[5] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 289–298.

[6] E. M. Nystrom, H.-S. Kim, and W. H. Wen-mei, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *International Static Analysis Symposium*. Springer, 2004, pp. 165–180.

[7] C. Lattner and V. Adve, "Data structure analysis: An efficient context-sensitive heap analysis," Tech. Report UIUCDCSR-2003-2340, Computer Science Dept., Univ. of Illinois . . . , Tech. Rep., 2003.

[8] L. Li, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 85–96, 2013.

[9] J. Whaley and M. Lam, "Context-sensitive pointer analysis using binary decision diagrams," Ph.D. dissertation, Citeseer, 2007.

[10] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 242–256, 1994.

[11] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[12] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 218–229.

[13] J. Zhu, "Towards scalable flow and context sensitive pointer analysis," in *Proceedings. 42nd Design Automation Conference, 2005.* IEEE, 2005, pp. 831–836.

[14] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam *et al.*, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM Sigplan Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[15] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[16] ——, "Value-flow-based demand-driven pointer analysis for C and C++," *IEEE Transactions on Software Engineering*, 2018.

[17] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," *ACM Sigplan Notices*, vol. 30, no. 6, pp. 1–12, 1995.

[18] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[19] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[20] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 290–299.

[21] LegUp Computing Inc., "LegUp 5.1 Documentation," 2017, https://www.legupcomputing.com/docs/legup-5.1-docs/index.html.

[22] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 2013, pp. 1–4.

[23] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis (v2018.2)*, 2018.

[24] L. Séméria and G. De Micheli, "SpC: synthesis of pointers in C: application of pointer analysis to the behavioral synthesis from C," in *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998, pp. 340–346.

[25] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013, pp. 362–365.

[26] F. Winterstein, *Separation Logic for High-level Synthesis*. Springer, 2017.

[27] N. Ramanathan, J. Wickerson, and G. A. Constantinides, "Scheduling weakly consistent C Concurrency for reconfigurable hardware," *IEEE Transactions on Computers*, vol. 67, no. 7, pp. 992–1006, 2017.

[28] N. Ramanathan, G. A. Constantinides, and J. Wickerson, "Concurrency-aware thread scheduling for high-level synthesis," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 101–108.

[29] N. V. Giamblanco and J. H. Anderson, "A dynamic memory allocation library for high-level synthesis," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 314–320.

[30] ——, "ASAP: Automatic sizing and partitioning for dynamic memory heaps in high-level synthesis," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 275–278.

[31] Z. Xue and D. B. Thomas, "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–7.

[32] ——, "SynADT: Dynamic data structures in high level synthesis," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 64–71.

[33] Y. Sui and J. Xue, "PTABen Benchmark suite," 2020, https://github.com/SVF-tools/PTABen.

[34] C. Lattner and V. Adve, "The LLVM instruction set and compilation strategy," *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.