A case for precise, fine-grained pointer synthesis in high-level synthesis

NADESH RAMANATHAN, Imperial College London, UK
GEORGE A. CONSTANTINIDES, Imperial College London, UK
JOHN WICKERSON, Imperial College London, UK

This article combines two practical approaches to improve pointer synthesis within HLS tools. Both approaches focus on inefficiencies in how HLS tools treat the *points-to graph* – a mapping that connects each instruction to the memory locations that it might access at runtime. HLS pointer synthesis first compute the points-to graph via pointer analysis and then implements its connections in hardware, which gives rise to two inefficiencies. Firstly, HLS tools typically favour pointer analysis that is fast, sacrificing precision. Secondly, they also favour centralising memory connections in hardware for instructions that can point to more than one location.

In this article, we demonstrate that a more precise pointer analysis coupled with decentralised memory connections in hardware can substantially reduce the unnecessary sharing of memory resources. We implement both flow- and context-sensitive pointer analysis and fine-grained memory connections in two modern HLS tools, LegUp and Vitis HLS. An evaluation on three benchmark suites, ranging from non-trivial pointer use to standard HLS benchmarks, indicates that when we improve both precision and granularity of pointer synthesis, on average, we can reduce area and latency by around 42% and 37% respectively.

ACM Reference Format:
Nadesh Ramanathan, George A. Constantinides, and John Wickerson. 2021. A case for precise, fine-grained pointer synthesis in high-level synthesis. 1, 1 (October 2021), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High-level synthesis (HLS) is the process of automatically compiling behavioral descriptions expressed in software, such as in C, into hardware expressed in register-transfer level (RTL) [8]. HLS is beginning to gain traction in industry since it improves design productivity and its quality of results are becomingly comparable to hand-written RTL [3, 18]. As such, it is crucial to keep pushing the boundaries of modern HLS compilers to support more program features and better analyses [25]. One such effort to push the boundaries of C-based HLS is the synthesis of pointers.

In C, pointers enable dynamic addressing of memory locations, which in turn enables the expression of dynamic data structures such as linked lists and trees. Although many HLS tools accept C as their input language, most do not have good support for pointers. For instance, some HLS tools, like LegUp [5] and Bambu HLS [27] are overly conservative with implementing pointers, while others, like Vitis HLS [47], do not support pointers at all. Due to the recent surge of interest in HLS, we believe that now is the time to scrutinise and explore efficient pointer synthesis within modern-day HLS tools. Furthermore, recent HLS works on synthesising pointer-manipulating

Authors' addresses: Nadesh Ramanathan, Imperial College London, South Kensington Campus, London, SW7 2AZ, UK; George A. Constantinides, Imperial College London, South Kensington Campus, London, SW7 2AZ, UK; John Wickerson, Imperial College London, South Kensington Campus, London, SW7 2AZ, UK.

XXXX-XXXX/2021/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

programs [44, 45], atomic pointers [30, 32] and dynamic memory allocation [10, 11, 24, 48, 49] are examples where pointers are used non-trivially, increasing the need for efficient pointer synthesis in future HLS tools.

The key to implementing pointers in HLS is the *points-to graph*, which associates each instruction in the program with the set of memory locations it might access at runtime [17, 36]. The standard HLS approach is first to compute the points-to graph and then to implement its connections between memory instructions and memory elements in hardware. In this article, we focus on inefficiencies that arise during these two steps.

Precise pointer analysis. The points-to graph of a program is computed by a *pointer analysis*. The number of connections in this graph can depend on the precision of this analysis. Although fully precise pointer analysis is undecidable [29], there are several analyses that can yield good precision within a reasonable time on large codebases [9, 13, 15, 40, 41, 43, 52].

Existing HLS tools tend to sacrifice precision in favour of quicker analysis times. For example, LegUp HLS uses an implementation of Andersen analysis [13], as it claims that the compiler community has developed fast insensitive analyses [21, §4.11]. Alas, Andersen analysis is flow- and context-insensitive. We demonstrate that switching to a pointer analysis due to Sui and Xue [39] that is flow-sensitive (considers the order of instructions in a program) and context-sensitive (considers a function's calling context) can substantially reduce false sharing of memory resources, with minimal impact on analysis times.

Fine-grained memory connections. Once the points-to graph is computed, HLS tools must determine how to connect instructions to the memory locations that they could point to during runtime. Existing HLS tools that support pointers take what we call a *coarse-grained* approach, where a centralised memory subsystem is put in place. Any instructions that can point to more than one memory location are forced to access these locations via a global memory controller. This approach ensures accessibility but introduces complicated circuitry for address arbitration and forces independent memory accesses to be serialised. LegUp takes this approach for all instructions, except for those that are connected to just one location in the points-to graph [22, §4.2.3.1]. Vitis HLS only synthesises programs where *all* memory instructions are connected to just one location [46, Chapter 1].

Instead of adopting coarse-grained memory connections, as provided by modern HLS tools, we investigate the efficacy of decentralising the memory connections between instructions and the locations they can point to at runtime. We implement fine-grained memory connections in which the global memory controller is eschewed in favour of a series of per-instruction arbiters. The idea is to connect each instruction directly to the memory locations that it might access at runtime (as determined by the pointer analysis), and to handle address disambiguation locally. Although there are well-known approaches to decentralise memory connections [33–35], they have largely been overlooked and unimplemented by modern HLS tools. However, given the rising influence of HLS and the growing need for pointers in HLS, such approaches must be revisited systematically. We must especially explore their effects within the context of also scrutinising points-to precision.

Article Outline. In summary, we make the case for improving the precision and granularity of pointer synthesis of HLS tools. We outline the following material in this article:

(1) In §2, we present a worked example that demonstrates why and how precision and granularity influence pointer synthesis within HLS.
(2) In §3, we describe how we have prototyped flow- and context-sensitive pointer analysis (§3.1) and fine-grained memory connections (§3.2) within two modern HLS tools, LegUp [21] and Xilinx Vitis HLS [47].

Table 1. A summary of how precise, fine-grained pointer synthesis affects area and latency. (More details about our experiments are in Section 4.) We write ▼ to indicate decreases and ▲ to indicate increases. On average, improving the precision and granularity of pointer analysis leads to 42% and 37% in area and latency reductions across three benchmarks and two HLS tools.

	LegUp HLS		Vitis HLS	
benchmark	area	latency	area	latency
PTABen	▼67%	▼49%	▼40%	▼2%
Array Partitioning	▼11%	▼16%	▼94%	▼92%
CHStone	▲1%	▼2%	-	-
combined	▼25%	▼22%	▼67%	▼47%

(3) In §4, we evaluate the effects of precision and granularity of pointer synthesis in both HLS tools using three benchmark suites:
- a benchmark suite with non-trivial pointer use called PTABen [50],
- a benchmark suite used for HLS array partitioning [6], and
- a standard HLS benchmark suite called CHStone [12].

Summary of results. The choice of benchmarks for evaluating our work is a delicate one. We have included the standard CHStone benchmark because it is widely used, but as shown in Table 1, our approach has a negligible effect there. (That said, we found that disabling function-inlining on these benchmarks leads to more opportunities for precise pointer synthesis, as we discuss in §4.3.) The problem with CHStone is that it is designed to reflect the kind of programs that work well with current HLS tools, and hence avoids non-trivial pointer use, probably because HLS tools do not support pointers very well. In turn, we believe that one reason HLS tools do not have good pointer support is the lack of demand for non-trivial pointer use from HLS benchmarks. Thus, in this article, we seek to move beyond this chicken-and-egg situation by including other benchmarks too.

The PTABen benchmarks involve non-trivial use of pointers, and Table 1 shows that precise, fine-grained pointer synthesis has the most impact on these benchmarks. Some improvements are also seen from the array partitioning benchmarks, which consist of more regular but partitioned memory accesses. Current HLS tools can do a decent job on these benchmarks but there are still inaccuracies in interpretation of the points-to graph.

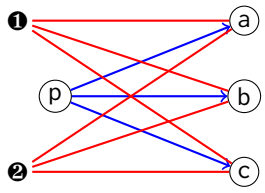
Comparison to our prior work. This article extends our conference paper at FPL 2020 [31] in three ways. Firstly, in our conference paper, we identified that the points-to graph supplied to HLS pointer synthesis is often imprecise. In this article, we further identify that pointer synthesis itself is centralised and does not completely exploit the reduction in memory connections made possible by precise pointer analysis. Hence, we explore a holistic approach of not only focusing on precision but also connection granularity to improve the efficiency of pointer synthesis in HLS. Secondly, where the conference paper only prototyped our implementations for the LegUp tool, this article shows how our implementations are also effective when applied to the Xilinx Vitis HLS tool. Finally, where the evaluation in our conference paper was limited to the PTABen benchmark suite, this article includes two further benchmark suites: array partitioning and CHStone.

Supplementary material. We provide some supplementary material which includes an LLVM pass that allows programs with pointers that can point to multiple locations, which would otherwise not be synthesisable via Vitis HLS [1]. The details of this LLVM pass are discussed in §3.2.2.

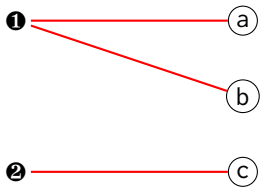
```
int a = 1, b = 2;
int c = 3, r = 0;
int *p;

void f() {
 r += *p; ❶
}
```

```
int main() {
 p = &a; f();
 p = &b; f();
 p = &c;
 r += *p; ❷
 return r;
}
```

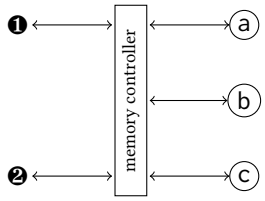
(a) a program



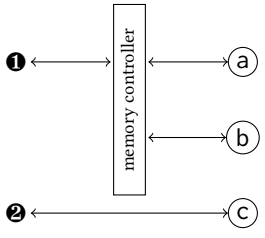
(b) imprecise analysis



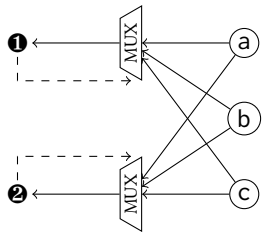
(c) precise analysis



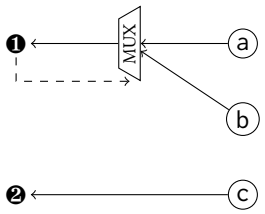
(d) imprecise coarse-grained hardware



(e) precise coarse-grained hardware



(f) imprecise fine-grained hardware



(g) precise fine-grained hardware

Fig. 1. An example program that uses pointers non-trivially. Figs. 1b and 1c shows the points-to graph generated by an insensitive Andersen analysis and precise analysis respectively. Figs. 1d and 1e represents the hardware generated by LegUp, using the results of Figs. 1b and 1c respectively. Figs. 1f and 1g represents hardware generated by our modifications to both LegUp and Vitis HLS, using the results of Figs. 1b and 1c respectively. In Fig. 1b, blue arrows are the results on Andersen analysis. Red lines are the points-to relation between pointer instructions and memory variables.

2 MOTIVATING EXAMPLE

In this section, we discuss an example that shows how precise analysis and fine-grained memory connections influence HLS-generated pointer hardware.

Consider the program in Fig. 1a, which consists of a pointer, p, and four variables, a, b and c and r. Overall, the program accumulates the dereferenced value of p three times. We disable inlining for this example, to avoid HLS optimisations. In this program, p is first assigned to a, then to b and finally to c. Note that the function f is called after the first two assignments. p is dereferenced once within function f, labelled as ❶, and once in main, labelled as ❷.

At the LLVM-level, each direct access to p is a LLVM instruction. Additionally, dereferencing pointer p requires two LLVM loads.[1] In practice, the code in Fig 1a also generates three stores to p and two loads of p. For brevity, we have eliminated these direct accesses to p in our diagrams.

Insensitive Andersen analysis. When Andersen analysis interprets this program, it infers that p may point to either a, b or c, as shown by the blue arrows in Fig. 1b. Notice that Andersen analysis relates variables without considering any instructions, i.e. the output of Andersen analysis is {(p, a), (p, b), (p, c)}. From this output, the best a HLS tool can infer is that both ❶ and ❷ may access a, b or c during runtime, i.e. {(❶, a), (❶, b), (❶, c), (❷, a), (❷, b), (❷, c)} , as shown by the red lines in Fig. 1b. In summary, Andersen analysis suggests that both memory instructions may access any of the three variables at runtime.

Some HLS tools, such as LegUp, enable such possibility via its addressable memory controller that handles pointer disambiguation at runtime. Fig. 1d shows hardware generated using a memory controller based on Andersen analysis. On one side, instructions ❶ and ❷ are connected to the controller and, on the other side, memory elements of a, b and c are also connected to the controller. Additionally, ❶ and ❷ may alias and must not be scheduled in parallel or out-of-order.

Precise pointer analysis. A flow- and context-sensitive analysis understands the order in which the assignment to p occur: a, b and then c. Hence, a precise analysis can infer that ❶ is called after p is assigned to a and b, but not c. Also, it infers that ❷ always accesses c. As a consequence, a precise analysis provides a more refined points-to result in Fig. 1c, compared to Andersen analysis in Fig. 1b. Notice that precise analysis relates instructions to variables, rather than variables to variables as in Andersen analysis.

This refined points-to graph influences pointer synthesis in HLS tools, reducing addressing circuitry by removing false sharing. For example, ❶ only accesses a and b, and ❷ only ever accesses c during runtime. Hence, only ❶, a and b need to be connected to the memory controller and ❷ can be directly connected to c, as shown in Fig. 1e. This optimisation reduces arbitration within the memory controller and simplifies the addressing circuitry of ❷ and c. Finally, the scheduler can reorder ❶ and ❷.

Fine-grained memory connections. Figs. 1d and 1e show groups of instructions and memory elements that have been forced to share a single physical connection. This physical connection limits the possibility of independent accesses and also increases false sharing. Instead using a memory controller, we can localise the memory connections on a per-instruction basis.

Fig. 1f shows how we introduce fine-grained memory connections, instead of the memory architecture in Fig. 1d. We introduce multiplexing per pointer load (or demultiplexing per pointer store) based on the set of inputs of the points-to graph. In this case, the points-to graph is from Andersen analysis, i.e. Fig. 1b. The selection criterion of each multiplexer is the runtime value of p. Using fine-grained connections simplifies the addressing circuitry in two ways. Firstly, the memory

[1] Since LLVM memory instructions use three-address code [19]

elements can be accessed without latency delays from the memory controller. Secondly, arbitration is localised to each instruction.

Combining precision and fine-grained connections. The key benefit of using fine-grained connections is that we can customize the memory connections for each pointer instruction. This customization is advantageous when the points-to graph is precise for each instruction, as in Fig. 1c. Fig. 1g shows the memory architecture resulting from improving both precision and connection granularity. ❶ is connected to a and b via a multiplexer, since the runtime value of p can only point to these two elements. ❷ is directly connected to c, without necessitating a multiplexer, or even loading the runtime value of p. In summary, this generated hardware is smaller and faster than all previous generated hardware discussed in this example.

3 METHOD

In this section, we prototype our implementations to improve precision and connection granularity of pointer synthesis. Our implementations focus on tackling inefficiencies both generation and interpretation of points-to graph within modern HLS tools.

In §3.1, we focus on introducing more precise pointer analysis when generating the points-to graph. Although precise pointer analysis is well studied in the software world, it has been hardly considered in the HLS context. Hence, we leverage flow- and context-sensitive pointer analysis from SVF [39], which is an LLVM-based pointer analysis tool. We can pass the intermediate LLVM IR of both LegUp and Vitis HLS to SVF to generate more precise points-to graphs. Handling these precise graphs require a fundamental shift in how modern HLS tools view points-to graphs. We discuss this view in detail.

Then, in §3.2, we focus on inferring fine-grained memory connections when interpreting the points-to graph. Although well-known methods [34, 35] exist, these methods have neither been considered in comparison to modern HLS memory controllers nor evaluated in the context of points-to precision of pointer synthesis. We explore both these aspects. We prototype two implementations within LegUp and Vitis HLS respectively to improve connection granularity. For LegUp HLS, we tap into its backend to prevent the generation of a global memory controller, but instead prototype memory arbitration on per-instruction basis. For Vitis HLS, we prototype a front-end implementation inspired by Séméria *et al.* [34, 35]. Both these prototypes are then supplied with refined points-to graphs from precise pointer analysis, to evaluate the influence of points-to precision within HLS.

3.1 Leveraging precise pointer analysis

To leverage precise analysis, we must first understand how Andersen analysis is used in HLS. Let V be the set of variables in the IR code. Andersen analysis produces a points-to relation between variables, $AnderPts \subseteq V \times V$. For example, $AnderPts = \{(\mathsf{p}, \mathsf{a}), (\mathsf{p}, \mathsf{b}), (\mathsf{p}, \mathsf{c})\}$ based on the IR code in Fig. 1a, as shown by the blue arrows in Fig. 1b.

3.1.1 Understanding imprecise analysis. An LLVM-based HLS tool uses the results of Andersen analysis to generate the memory addressing for all indirectly-addressed LLVM memory instructions. From this analysis, an HLS tool infers a points-to relation between instructions and variables. Let I be the set of LLVM memory instructions. Let $deref \subseteq I \times V$ relate instructions to pointers that it dereferences. *InstPts* relates instructions and variables, i.e. $InstPts \subseteq I \times V$. *InstPts* is inferred using the results of Andersen analysis, as follows:

$$InstPts = DirectPts \cup IndirectPts$$

where

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

DirectPts represents directly addressed instructions, which are instructions whose variables are expressed in the LLVM source. For example, all loads and stores to `@p` and `@r` in Fig. 1a are part of *DirectPts*. *IndirectPts* represents indirectly addressed instructions, where this relation between instructions and variables that must be inferred from Andersen analysis. *IndirectPts* defines that an instruction i points to variable v if instruction i dereferences a pointer v_p, i.e. $(i, v_p) \in deref$, and Andersen analysis states that v_p points to v. For example, since $AnderPts = \{(\mathsf{p}, \mathsf{a}), (\mathsf{p}, \mathsf{b}), (\mathsf{p}, \mathsf{c})\}$ and $\{(❶, \mathsf{p}), (❷, \mathsf{p})\} \in deref$, ❶ and ❷ must be related to a, b and c i.e. $IndirectPts = \{(❶, \mathsf{a}), (❶, \mathsf{b}), (❶, \mathsf{c}), (❷, \mathsf{a}), (❷, \mathsf{b}), (❷, \mathsf{c})\}$. Together, *DirectPts* and *IndirectPts* form a points-to graph that LegUp uses for memory addressing. *IndirectPts* is shown by red edges in Fig. 1b.

3.1.2 Implementing precise analysis. A precise pointer analysis produces a points-to relation between instructions and variables, i.e. $I \times V$, for all LLVM memory instructions. We utilise the SVF pointer analysis [39], that can be configured as either a flow-sensitive analysis ($FSInstPts \subseteq I \times V$) or a flow-and-context-sensitive analysis ($FSCSInstPts \subseteq I \times V$). In general, $FSCSInstPts \subseteq FSInstPts \subseteq InstPts$, 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.

An LLVM-based HLS tool can directly use the points-to relation of SVF for memory addressing, i.e. $IndirectPts = FSInstPts$ or $IndirectPts = FSCSInstPts$. This is because SVF directly provides the relation between instructions and variables ($I \times V$) to the HLS tool, rather than forcing the HLS tool to derive the points-to relation from Andersen analysis that only relates variables ($V \times V$). For example, for the code in Fig. 1a, SVF generates $IndirectPts = FSInstPts = \{(❶, \mathsf{a}), (❶, \mathsf{b}), (❷, \mathsf{c})\}$. SVF encodes that ❶ only accesses a and b and ❷ only accesses c, as seen in Fig. 1c.

3.1.3 SVF implementation details. We did not tightly integrate LegUp and Vitis HLS with SVF, since all three tools were implemented with different LLVM versions. So, we designed a simple interface between these HLS tools and SVF via file I/O streams. SVF is a demand-driven pointer analysis [16], which applies flow- and/or context-sensitive analysis on a per-instruction basis. However, we configure SVF to analyse the entire program, with unlimited time and memory budgets, since we want to exploit the best-case precision for every instruction. We configure SVF to support several LLVM instructions for indirect accesses including loads, local variables, arrays, function arguments, selects and phi-nodes. We can also support multiple pointer indirections, since it still translates to instructions points to variables.

3.2 Introducing fine-grained memory connections

Whenever an instruction points to more than one variable, the HLS-generated hardware must be able to cope with pointer disambiguation at runtime. The set of instructions that require pointer disambiguation, *MultiAddrInst*, is defined as follows:

$$MultiAddrInst = \{i \in I \mid \exists v \in V.\, \exists v' \in V.\, v \neq v' \wedge (i, v) \in InstPts \wedge (i, v') \in InstPts\}$$

where an instruction i is in *MultiAddrInst* if it points to at least two distinct variables v and v' in *InstPts*. In the next two subsubsections, we discuss how LegUp and Vitis HLS natively deal with pointer instructions that can point to more than one variable and how we introduce per-instruction arbitration for these instructions.

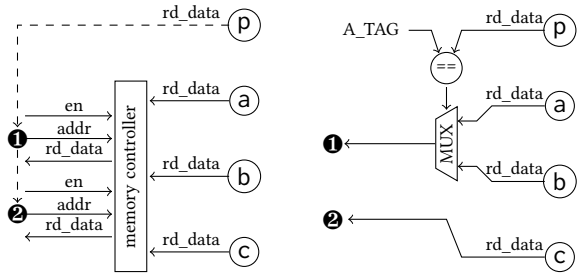


Fig. 2. Implementing indirect loads of code in Fig. 1a within LegUp HLS.

3.2.1 Generating per-instruction arbitration without memory controller in LegUp HLS. LegUp HLS supports pointer disambiguation via a global memory controller. All the instructions in *MultiAddrInst* are connected to one side of memory controller. Any memory variables can be accessed by *MultiAddrInst* must be also be connected to the controller, which LegUp refers to as global memories. The set of LegUp's global memories, *GlobalMem*, is defined as follows:

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

where any variable v is implement as global memory (*GlobalMem*) if an instruction i is in *MultiAddrInst* and points to v. All other variables are implemented in local memories, which can be directly connected.

For example, LegUp infers the following from Andersen analysis in Fig. 1b: *MultiAddrInst* = {❶, ❷}, *GlobalMem* = {a, b, c} and *LocalMem* = ∅. The memory connections in Fig. 1d is based on this imprecise inference. However, LegUp infers the following from SVF's precise analysis in Fig. 1c: *MultiAddrInst* = {❶}, *GlobalMem* = {a, b} and *LocalMem* = {c}. The memory connections in Fig. 1e are based on this precise inference.

Fig. 2a shows the original LegUp connections to handle pointer disambiguation. Each instruction must obey the signalling protocol of the controller: enable, write enable, address, read data and write data signals. Loads only require three signals as seen in the figure, whereas stores require four signals. If the memory element is an array, then the address signal is also required to index the array. In this example, the address signal is provided to ❶ and ❷ by the runtime value of p, as shown in the dotted lines of Fig. 2a. This value is directly wired as the address signal to the memory controller.

Fig. 2b shows how we can customise each instruction's memory connections based on the points-to graph in Fig. 1c. For ❶, we read the value of p and compare it against the address tag of a, i.e. A_TAG. We maintain the same addressing space from the memory controller for the comparison. Then, we feed the result of the comparison to a multiplexer that is tailored to ❶. Based on the points-to graph, the choice for ❶ is only between a and b. Additionally, ❷ is directly wired to c. We do not even need to read the runtime value of p for ❷ in Fig. 2b, compared to Fig. 2a. In general, we must introduce multiplexing for as many variables that an instruction can point to and for all necessary signals depending on whether the instruction is a load or a store.

3.2.2 LLVM transformations to support fine-grained connections in Vitis HLS. Natively, Vitis HLS does not provide any hardware to perform address disambiguation at runtime. As such, Vitis HLS

Before	`@a = global i32 1`
	`@b = global i32 2`
	`@c = global i32 3`
	`@p = global i32* NULL`

After	`@a = global i32 1 // A_TAG=1`
	`@b = global i32 2 // B_TAG=2`
	`@c = global i32 3 // C_TAG=3`
	`@p = global i32 0 // P_TAG=4`

(a) enumerating address

Before	`store @a, ** @p`
	`store @b, ** @p`
	`store @c, ** @p`

After	`store 1, @p // A_TAG=1`
	`store 2, @p // B_TAG=2`
	`store 3, @p // C_TAG=3`

(b) updating pointer addresses

Before	`%1 = load ** @p` `%2 = load * %1` ❷

After	`%1 = load @p`
	`%2 = load @a`
	`%3 = load @b`
	`%4 = icmp %1, 1 // A_TAG=1`
	`%5 = select %4, %2, %3`
	`%6 = load @c`
	`%7 = icmp %1, 3 // C_TAG=3`
	`%8 = select %7, %6, %5` ❶

(c) replacing indirect load of ❷ after Andersen analysis

Before	`%1 = load ** @p` `%2 = load * %1` ❷
After	`%2 = load @c` ❷

(d) replacing indirect load of ❷ after precise analysis

Fig. 3. Snippets of LLVM IR from Fig. 1a before and after our Vitis transformation, inspired by Séméria *et al.* [34, 35].

is fundamentally limited in its ability to support pointer-manipulating code [46, Chapter 1]. Unless the pointer analysis can statically determine the single location that a pointer dereferences from, Vitis HLS forbids the pointer code from being synthesised, i.e. it only supports pointer code when $MultiAddrInst = \emptyset$.

In order to evaluate any non-trivial pointer code in meaningful manner, we have to address this limitation within Vitis HLS. Hence, we extend Vitis HLS to handle cases when $MultiAddrInst \neq \emptyset$. As mentioned earlier, Vitis HLS does not provide any memory controller capabilities. We considered

introducing a memory controller in hardware at Verilog level for Vitis HLS. Unfortunately, this option was ruled out since we had no access to its back-end generator, unlike with LegUp HLS. Therefore, the possibility of implementing coarse-grained memory connections was eliminated within Vitis HLS. As consequence, we focussed on introducing a front-end optimisation to support pointer synthesis. Vitis HLS now enables us to tap into its front-end by injecting an LLVM source-to-source pass. Using this mechanism, we introduce fine-grained memory connections for pointer synthesis within Vitis HLS, which we provide in our supplementary material.

We implement a standard HLS method to implement fine-grained memory connections by Séméria *et al.* [34, 35]. Although this method is well-known, it has not been implemented in the context of evaluating precision of pointer synthesis, which is the focus of this section. We re-write the LLVM IR compiled from Fig. 1a, which involves three key transformations, as illustrated in Fig. 3.

The first transformation enumerates all memory locations in the code to allow direct addressing of these variables, as shown in Fig. 3a. We also replace pointer variables with regular variables, just like the declaration of `@p`. Our pass only supports global variables and arrays currently. We can extend our prototype to support local variables in the future.

The second transformation replaces all direct pointer instructions. For example, Fig. 3b shows three stores to `**p` that we replace with the respective address tags. These transformations must be implemented to both loads and stores. We must also keep track of all the replaced instructions and make sure we replace its use within each basic block.

The third transformation replaces all pointer dereferencing instructions with direct loads or stores and selection (multiplexing) mechanism based on the enumerated addresses. These injected instructions are designed to mimic the same behaviour as the replaced pointer instruction. The set of direct loads and stores are directly obtained from the points-to graph, which we use to load, compare and select the correct variable based on the runtime value of the pointer.

This is where points-to precision matters. If we inject instructions based on Andersen analysis, as in Fig. 1b, then we have to inject four loads (`p`, `a`, `b` and `c`) and two compare and select pairs to replace ❷, as shown in Fig. 3c. This is because Andersen analysis deems that ❷ could point to `a`, `b` or `c` at runtime. On the other hand, precise analysis eliminates two of three possibilities and states with certainty that ❷ only ever accesses `c`. Consequently, we can simply inject one direct load from `c` to replace ❷ with precise analysis, as shown in Fig. 3d.

Hence, the quality of pointer analysis can directly influenced the number of injected instructions, which affects operation scheduling (due to aliasing) and circuit area as well. We also handle indirect stores, which is similar to how we handle indirect loads, only we have to also read from and then update all the locations that a pointer could point to.

More details on our transformation can be obtained from our LLVM source code, which is presented at [1].

4 EVALUATION

We evaluate our prototype implementations on pointer synthesis on three benchmark suites and two different HLS tools. The variation of benchmarks are designed to represent the program properties that range from standard to challenging programs for current HLS tools. The variation in HLS tools shows that our implementations are generally applicable to a variety of LLVM-based HLS tools.

Design points. Fig. 4 shows our six design points: four and two within LegUp and Vitis HLS respectively. The rightward arrows represent our implementation that moves from an imprecise pointer analysis like Andersen analysis to a precise pointer analysis within both HLS tools, as

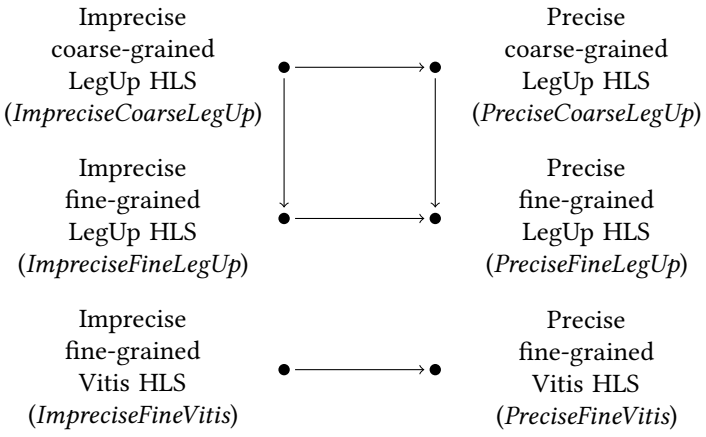


Fig. 4. Design points for evaluation.

discussed in §3.1. The downward arrows represent our implementation that introduces fine-grained memory connections within LegUp HLS, as discussed in §3.2.1. We also extend Vitis HLS to directly provide fine-grained connections, as discussed in §3.2.2. Note that Vitis HLS does not provide any coarse-grained memory connections natively.

Software and synthesis setup. We use different synthesis setups per HLS tool. We also do not directly compare between HLS tools, since their HLS optimisations and stages vary. For LegUp HLS, we use LegUp 5.1's pure hardware, which synthesises all memories in a C program as FPGA registers or RAMs. We synthesise and place-and-route all designs using Quartus v15.0 to a Cyclone V SoC FPGA (5CSEMA5F31C6N). For Vitis HLS, we use Vitis HLS v20.1. We synthesise and place-and-route all designs using Vivado v20.1 to a Xilinx Kintex 7 FPGA (xc7k160tfbg484-1).

Section outline. This section is outlined as follows. In §4.1, we evaluate our prototypes on the PTABen benchmark suite. In §4.2, we evaluate our prototypes on the array partitioning benchmark suite. In §4.3, we evaluate our prototypes on the CHStone benchmark suite. In §4.4, we discuss the impact of precise analysis on the analysis times via a pathological example. In §4.5, we summarise our implementations on these benchmarks across HLS tools.

4.1 Evaluation of PTABen programs

What is PTABen? The PTABen benchmark suite comprises over 400 hand-written programs that test for correctness and precision of pointer analyses. Although these pointer-based programs may be common programming patterns within modern software codebases, these patterns are relatively new to the HLS community.

Which programs do we select? PTABen organises its programs in terms of testing objectives. We identified programs whose objective is to test the flow- and context-sensitivity of pointer analysis (two subfolders). Out of 50 programs, we are able to synthesise 32 programs. The remaining programs are unsynthesisable because they require dynamic memory allocation, recursion or C structures.

Do we make changes to these programs? We minimally modify these 32 programs from PTABen for our purposes. PTABen inserts alias checks, via backdoor calls to SVF, that are meant to instrument the points-to precision of various instructions. We convert these backdoor calls into non-inlined

Program	Memory instructions	Memory locations
cs0	25	8
cs1	29	11
cs2	56	23
cs3	53	20
cs4	38	16
cs5	38	8
cs6	39	7
cs7	34	12
cs8	34	17
cs9	44	19
cs10	46	15
cs11	22	8
cs12	22	16
cs13	32	9
cs14	24	9
cs15	14	14
cs17	68	20
cs18	8	13
cs19	18	11
cs20	29	17
simple1	18	6
simple2	26	9
simple3	18	9
global1	15	8
global2	15	8
global3	20	9
global4	34	10
global5	18	12
branch1	22	7
branch2	17	6
branch3	34	9
strong-update	12	12

Table 2. Number of memory instructions and locations per PTABen program

functions where we dereference all pointers involved. These changes enable us to instrument the points-to sets of individual instructions of HLS-generated hardware. We also accumulate all dereferenced values and return the final output, so that the HLS tools cannot optimise away any pointer-relation instructions.

What is the pointer profile of selected programs? Table 2 shows the number of memory instructions and memory locations within each PTABen program that we synthesised. These include both direct and indirect accesses as well as pointer and variable/array locations. On average, each PTABen program has a mean of 32 instructions (with standard deviation of 21 instructions) and 12 locations (with standard deviation of 5 locations).

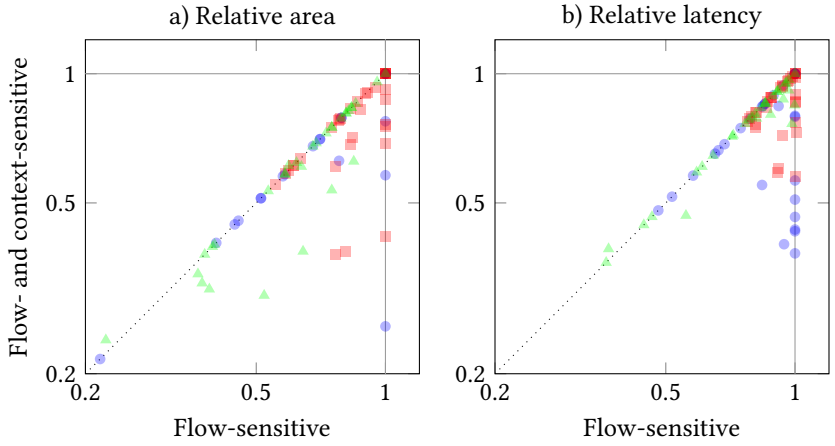


Fig. 5. Relative area and latency of different types of precise analysis, relative to imprecise Andersen analysis. Data points represent coarse-grained LegUp (●), fine-grained LegUp (■), and fine-grained Vitis (▲).

As a comparison, the CHStone benchmarks have a mean of 117 instructions (with standard deviation of 112 instructions), which is much larger and more varied than the PTABen programs. However, these consist of mostly direct accesses, and are a very small portion of the design. On the original LegUp implementation (*ImpreciseCoarseLegUp*), the CHStone programs average at 0.008 memory instructions per ALM, whereas the PTABen programs average at 0.1 memory instructions per ALM. Hence, the PTABen programs have a ratio that is at least an order of magnitude higher than CHStone, which shows the significant increase in complexity of pointer use, and highlights that the PTABen suite is moving beyond standard memory patterns synthesised by most HLS tools.

Result outline. First, we discuss the HLS effects of the different precise analyses that are possible via SVF in §4.1.1. Then, we present the LegUp and Vitis synthesis results in §4.1.2 and §4.1.3.

4.1.1 Understanding the impact of flow- vs flow-and-context-sensitive analysis on PTABen. SVF provides three different configurations: insensitive Andersen analysis, flow-sensitive analysis and flow-and-context-sensitive analysis. Hence, we have two choices when it comes to precise analysis. Fig. 5 shows the hardware area and latency of the two possible choices, relative to imprecise Andersen analysis. These graphs show us two important points. Firstly, regardless of tool or connection granularity, all points are in the lower-left quadrant. This means that both of the precise analyses always lead to improvements (or no change) in both area and latency, compared to imprecise Andersen analysis. Secondly, all the points are on or below the diagonal. This means that flow-and-context-sensitive analysis always leads to improvements (or no change) in both area and latency, compared to just flow-sensitive analysis. This trend is expected, since $FSCSInstPts \subseteq FSInstPts \subseteq InstPts$.

On average, hardware generated from flow- and context-sensitive analysis is 13% smaller and 10% faster than flow-sensitive analysis. Henceforth, when we mention precise analysis, we are referring to flow-and-context-sensitive analysis. All the rightward arrows in Fig. 4 are moving from Andersen analysis (imprecise analysis) to flow-and-context-sensitive analysis (precise analysis).

4.1.2 Evaluating precise, fine-grained pointer analysis of LegUp HLS on PTABen. Fig. 6 shows four metrics that we measure when synthesising the 32 programs from PTABen on our four LegUp-based design points in Fig. 4.

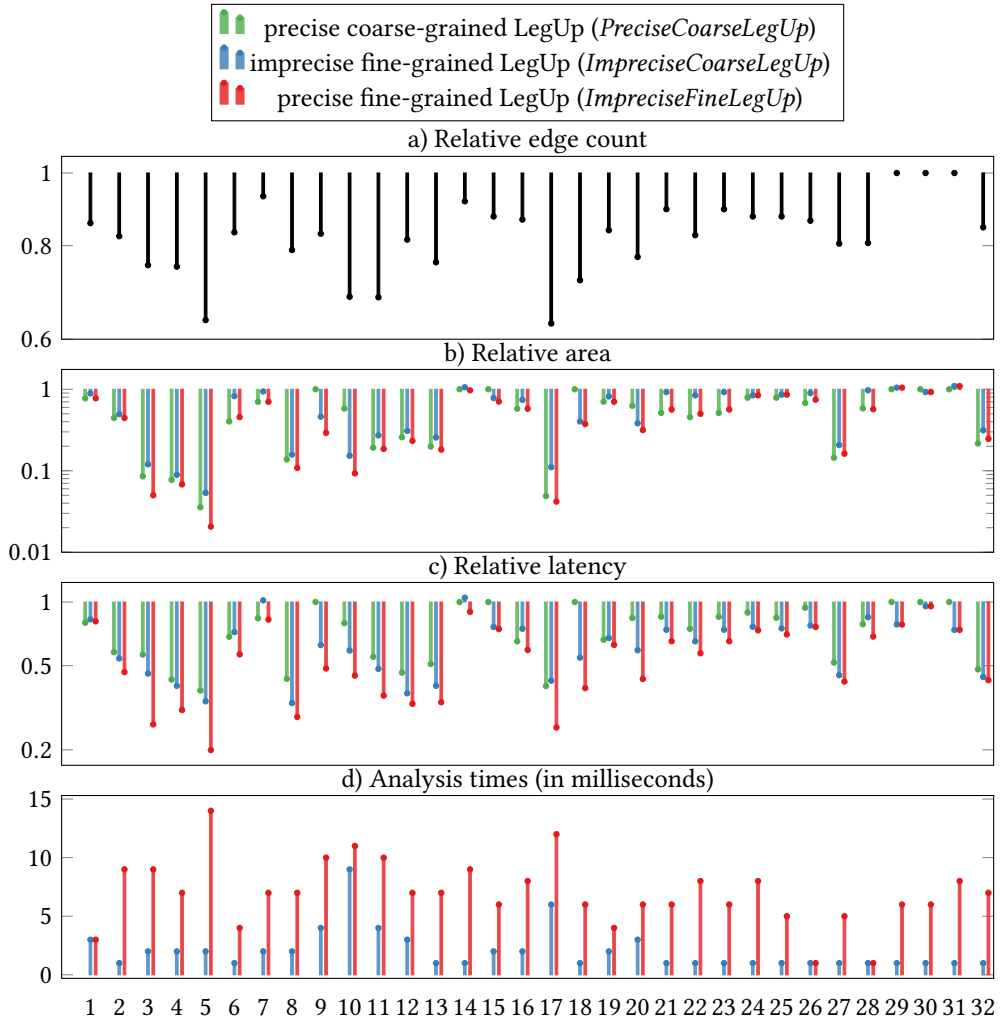


Fig. 6. Pointer synthesis metrics of LegUp by PTAben programs and by design point. We enumerate the following PTABen programs (from `cs_tests` and `fs_tests`) in this order: `cs0-15.c`, `cs17-20.c`, `simple1-3.c`, `global1-5.c`, `branch1-3.c` and `strong-update.c`. All metrics are relative to imprecise coarse-grained LegUp (*ImpreciseCoarseLegUp*).

The effects of precise analysis on points-to ratio. Fig. 6(a) shows relative edge count of the points-to graph generated by SVF using precise analysis, compared to imprecise Andersen analysis. On average, precise analysis reduces the relative edge count by 17%, compared to imprecise Andersen analysis. We also see that precise analysis can reduce relative edge count by a maximum 37%. There are also programs that do not benefit from precise analysis, i.e. the relative edge count is close to 1, which are the branch programs since sometimes control paths are hard to analyse even by precise analysis.

We only report a single bar per program, because both rightward arrows in Fig. 4 produce the same ratio, i.e. edge count of *PreciseCoarseLegUp* divided by edge count of *ImpreciseCoarseLegUp* is the same as edge count of *PreciseFineLegUp* divided by edge count of *ImpreciseFineLegUp*.

The effects of precise analysis on analysis times. These edge count reductions comes at a cost of additional time for pointer analysis. Fig. 6(d) shows the analysis times required for both imprecise and precise analysis. Although, on average, precise analysis is 4× slower than imprecise analysis, these analysis times, measured in milliseconds, are insignificant compared to the HLS compilation or hardware synthesis times. For perspective, it takes 5 seconds to compile Program 1 from C to Verilog and another 2 minutes to synthesise the generated Verilog into a bitstream within LegUp HLS.

The effects of precise, fine-grained pointer analysis on hardware runtimes and area. Fig. 6(b) and (c) show the relative area and latency of the generated hardware for our three LegUp-based design points relative to LegUp's original implementation, which is *ImpreciseCoarseLegUp*. Please note the area numbers in this paper refer to either ALM in LegUp HLS or LUT count in Vitis HLS, post-place-and-route. These numbers are implicitly proportional to multiplexing circuitry required to implement the memory sharing between pointer instructions and memory locations in hardware.

Introducing precise analysis. The first option is simply to provide LegUp with a more precise points-to graph generated by SVF's precise analysis, i.e. *PreciseCoarseLegUp*. On average, *PreciseCoarseLegUp* reduces area and latency by 60% and 30%, compared to *ImpreciseCoarseLegUp*. Hence, we see that reduction in edge count directly influences the quality of generated hardware.

Although the edge count is reduced for all except three programs (programs 29 to 31), we do not see reduction in area and latency for the same number of programs. For example, we do not see area reduction via *PreciseCoarseLegUp* in six programs (instead of three): Program 9, 14, 15, 18, 24 and 25. We also do not see latency reduction for programs 9, 14, 15 and 18. The reason for this is, despite the points-to graph becoming more precise, LegUp is unable to simplify the addressing circuitry or memory allocation, i.e. LegUp still uses their global memory controller. Despite $FSCSInstPts \subset InstPts$ for these programs, LegUp may generate the same *GlobalMem*. These observations led to our second option.

Introducing fine-grained connections. The second option is to use imprecise Andersen analysis but introduce fine-grained per-instruction arbitration, i.e. *ImpreciseFineLegUp*. On average, *ImpreciseFineLegUp* reduces area and latency by 54% and 39%, compared to *ImpreciseCoarseLegUp*, despite both design points using the same input points-to graph. *ImpreciseFineLegUp* can reduce area and latency by up to 96% and 67%, compared to *ImpreciseCoarseLegUp*. We see that fine-grained connections either reduce area or latency for all expect one program (Program 30). Avoiding use of the memory controller can result in reduced addressing circuitry or reduced cycle count to access the same memory element.[2]

On average, compared to *PreciseCoarseLegUp*, using fine-grained connections increases area but reduces latency. However, with closer observation, we see that whether *ImpreciseFineLegUp* or *PreciseCoarseLegUp* performs better depends on the program, as they can outperform each other. Sometimes reducing the edge count improves hardware performance more than fine-grained connections, and vice versa. Also, occasionally, *PreciseCoarseLegUp* is slightly above 1. These cases are rare and shows that sometimes *PreciseCoarseLegUp* may suffer from longer critical paths. These observations led to our third option.

Introducing both precise analysis and fine-grained connections. The third option combines precise analysis and fine-grained memory connections within LegUp, i.e. *PreciseFineLegUp*. On average, *PreciseFineLegUp* reduces area and latency by 67% and 49%, compared to *ImpreciseCoarseLegUp*.

[2]A load via a memory controller always takes one cycle, whereas direct load can be instantaneous, if it is a register. A store via a memory controller takes two cycles whereas a direct store takes one cycle.

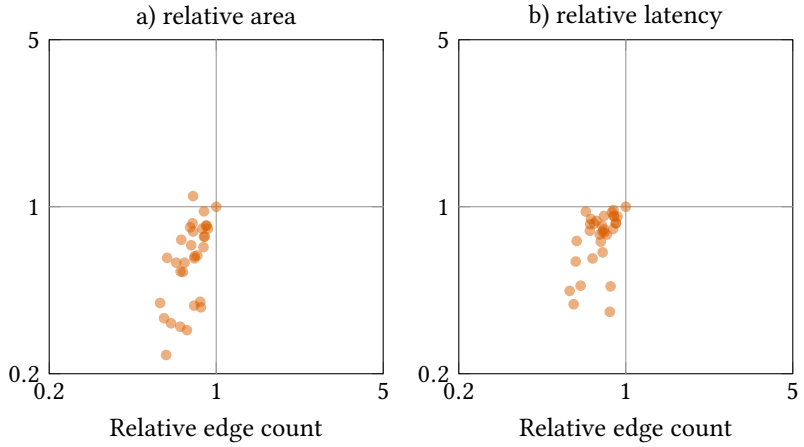


Fig. 7. Pointer synthesis metrics of Vitis for the same set of PTABen programs as in Fig. 6. All data points are relative to imprecise fine-grained Vitis (*ImpreciseFineLegUp*).

Hardware generated by *PreciseFineLegUp* can reduce area and latency by up to 98% and 80%. In general, we see that the benefits of precise analysis is amplified when memory generation is sensitive to the points-to precision. This is because *PreciseFineLegUp* reduces false sharing both at the level of graph connections as well as hardware memory connections.

4.1.3 Evaluating precise, fine-grained pointer analysis of Vitis HLS on PTABen. Fig. 7 shows relative area and latency versus relative edge count from employing precise analysis within Vitis HLS. As noted previously, Vitis HLS does not implement coarse-grained or fine-grained memory connections. We implement fine-grained memory connections within Vitis HLS, as discussed in §3.2.2. Then, we can apply precise analysis on top of our implementation of fine-grained connections in Vitis, as shown by rightward arrow in Fig. 4.

Firstly, note that almost all of the points are below $y = 1$. This means that employing precise analysis almost always leads to a reduction (or no change) in both area and latency. Also, on average, we see that employing precise analysis within Vitis HLS via *PreciseFineVitis* reduces area and latency by 40% and 26%, compared to *ImpreciseFineVitis*. *PreciseFineVitis* can reduce area and latency by up to 76% and 63%, compared to *ImpreciseFineVitis*. Only one program is 5% larger in *PreciseFineVitis*, compared to *ImpreciseFineVitis*, which we can attribute to a very small program having relatively large synthesis variations.

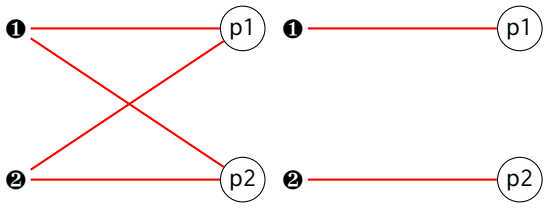
4.2 Evaluation of array partitioning benchmarks

What is EASY?. The next set of programs we evaluate are the array partitioning benchmarks from EASY by Cheng *et al.* [6]. These are a set of programs that consist of independent functions that are meant to access independent matrices that have been partitioned from the same source. As such, these partitioned programs are more typical benchmarks for HLS since consists of array accesses and loops with fixed intervals. We synthesise all five programs of this benchmark: *matrix multiplication*, *matrix addition*, *histogram*, *line-of-sight* and *substring*.

What is the programming paradigm of EASY?. For each benchmark, we partition the function and data by powers of 2, from 1 up to 16, and generate individual programs for each *partition count*. Figure 8 shows this programming paradigm, where we have two partitioned arrays and two

```
int *p;
int p1[SIZE], p2[SIZE];

int f1(){ return p[0];❶}

int f2(){ return p[0];❷}

int main(){
 int ret;
 p=p1; ret = f1();
 p=p2; ret += f2();
 return ret;
}
```

(a) an EASY program



(b) imprecise analysis (c) precise analysis

Fig. 8. A minimal example of the EASY programming paradigm (partition count of 2) and why precise analysis benefits pointer synthesis of both LegUp and Vitis HLS

independent functions sharing a pointer p. The main function assigns different array partitions to different function calls. Within each function, this pointer is dereferenced and used for the particular EASY computations. In the original EASY programs, these functions are compiled using C pthreads as independent functions, however the scope of this article is only for sequential C programs. We target these programs on both LegUp and Vitis HLS.

Figures 8b and 8c shows the points-to results of Andersen (imprecise) and precise analysis respectively. In the case of imprecise analysis, the HLS tool is informed that both functions could access both array partitions, which means the HLS tool must put in place pointer disambiguation circuitry. Precise analysis, on the other hand, identifies the one-to-one connection between the different functions and partitions, enabling simpler memory circuitry and reduces false sharing of independent partitions. In the case of Vitis HLS, the tool also identifies that there is no aliasing between `f1` and `f2`, and therefore the functions can run in parallel. In contrast, LegUp HLS executes one basic block at a time, but the memory arbitration is simplified by precise analysis.

4.2.1 Evaluating precise, fine-grained pointer synthesis of LegUp HLS on EASY. LegUp HLS only supports execution of a single basic block at any given time. Although functions are deemed to be independent, they cannot execute in parallel within LegUp HLS. Hence, the cycle count is very similar across partition counts and design points for each benchmark. So, as shown in Fig. 9, we report the clock frequencies versus area utilisation for each benchmark and LegUp design point (Fig. 4), where each data point is a partition count.

ImpreciseCoarseLegUp, the original LegUp implementation, suffers most clock frequency degradation as we increase the partition count for all five benchmarks. This design point connects all pointer instructions to all partitioned arrays, generating large fan-ins and fan-outs that directly delay the critical path.

Instead of connecting all instructions to global memory controller, we can introduce arbitration on a per-instruction basis, i.e. *ImpreciseFineLegUp* (downward arrow in Fig 4). The clock frequency of *ImpreciseFineLegUp* degrades at a slower rate than *ImpreciseCoarseLegUp*. On average, *ImpreciseFineLegUp* has 2% higher frequency than *ImpreciseCoarseLegUp*. *ImpreciseFineLegUp*'s frequency can either be up to 20% faster or slower than *ImpreciseCoarseLegUp*. This difference of degradation

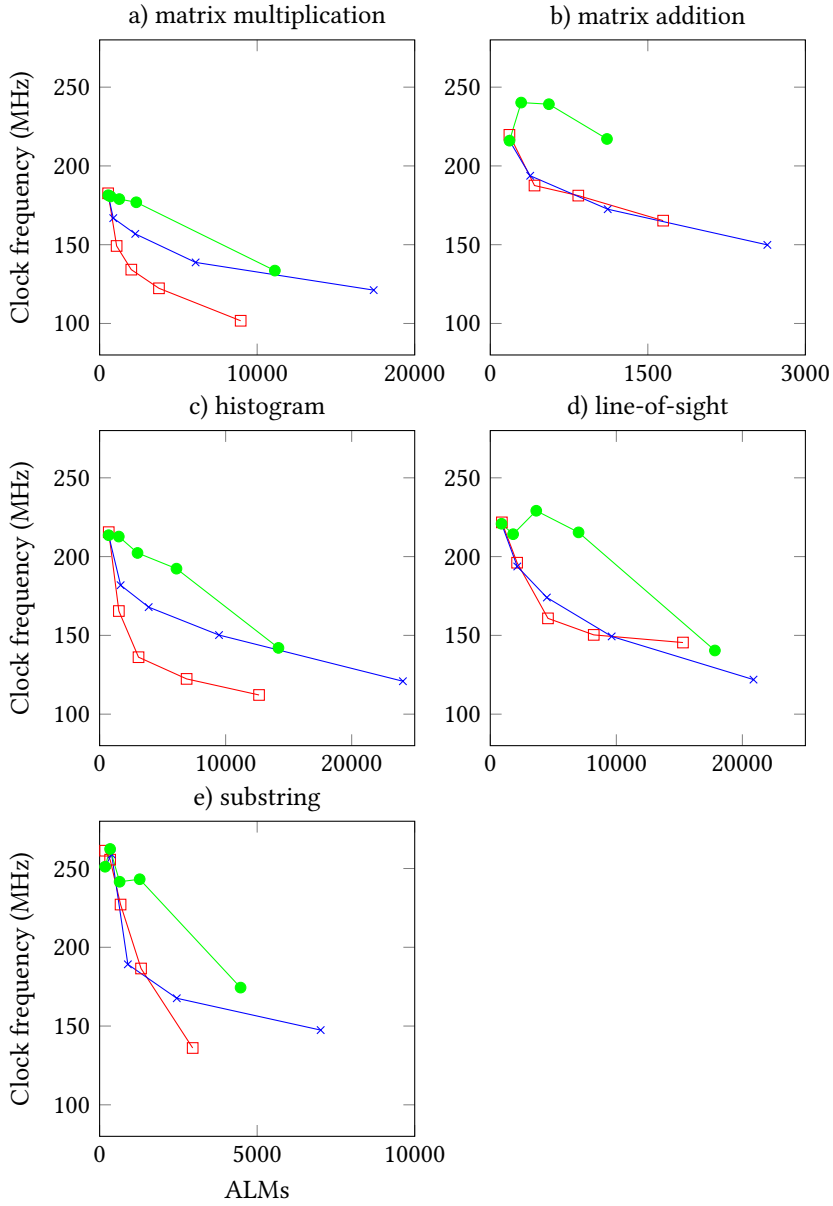


Fig. 9. Clock frequency against area utilisation in LegUp HLS for both imprecise and precise analyses on HLS benchmarks exploring array partitioning by Cheng *et al.* [6]. We show imprecise coarse-grained LegUp (*ImpreciseCoarseLegUp*, —□—), imprecise fine-grained LegUp (*ImpreciseFineLegUp*, —×—), precise LegUp (*PreciseCoarseLegUp* and *PreciseFineLegUp*, —●—)

is not very obvious for line-of-sight and substring, since their independent functions still share a few arrays.

What *ImpreciseFineLegUp* may gain in frequency, compared to *ImpreciseCoarseLegUp*, it loses in area overheads. Per-instruction arbitration is costly for arrays and gets costlier as we scale

the partition count. On average, *ImpreciseFineLegUp* generates circuits that are 30% larger than *ImpreciseCoarseLegUp*. *ImpreciseFineLegUp*'s hardware can be up to 2.3× larger than *ImpreciseCoarseLegUp*.

We also can employ precise analysis (rightward arrows in Fig 4) to these programs. Since precise analysis generates a points-to ratio of one for all these programs, the connection granularity does not matter. As such, both *PreciseCoarseLegUp* and *PreciseFineLegUp* generate the same hardware for these programs. On average, *PreciseFineLegUp* generates circuits that are 20% faster and 11% smaller, compared to *ImpreciseCoarseLegUp*. As we scale the partition count, *PreciseFineLegUp* can be up to 60% faster and 38% smaller compared to *ImpreciseCoarseLegUp*. *PreciseFineLegUp* is also 20% faster and 30% smaller, compared to *ImpreciseFineLegUp*. Reducing the number of points-to graph connections significantly reduces the area required for memory connections in hardware.

4.2.2 Evaluating precise, fine-grained pointer synthesis of Vitis HLS on EASY. Fig. 10 shows the impact of employing precise analysis within Vitis HLS. Vitis HLS allows for multiple functions to run in parallel using the `dataflow` pragma. However, this pragma only works if the alias analysis identifies that functions are independent. Precise analysis is able to do so (rightward arrow in Fig 4). On average, *PreciseFineVitis* is 3.7× faster and 38% smaller than *ImpreciseFineVitis*, with a maximum of 50× faster and 94% smaller. *PreciseFineLegUp* can be 70% slower and 50% larger than *ImpreciseFineVitis*, but only when the partition count is one. We see that the latency of matrix multiplication, matrix addition and histogram scales with partition count, unlike line-of-sight and substring. This is because the latter two benchmarks still share arrays after partitioning, which means the `dataflow` pragma does not parallelise functions.

4.3 Evaluation of CHStone benchmarks

What is CHStone and why did we choose it? The final set of benchmarks we evaluate is the CHStone benchmark suite [12]. CHStone is a standard HLS benchmark consisting of 12 programs from various domains such as arithmetic, media processing, security and microprocessor design. The main motivation for experimentation on the CHStone benchmark is to evaluate the performance of our methods on a standard HLS benchmark. We also chose CHStone since it is reflection of the argument that current HLS benchmarks only have trivial pointer use.

Which method and HLS tool did we evaluate CHStone on? There were two limiting factors in our experimentation specific to CHStone. Firstly, we do not discuss the impact of precise analysis on CHStone programs, i.e. all horizontal arrows in Figure 4. Since the pointer use is trivial, the points-to results of imprecise Andersen analysis and precise SVF analysis are the same. Therefore, we implemented precise analysis by default for CHStone. Instead, we focus on only understanding the effect of fine-grained memory connections on pointer synthesis.

Secondly, we only experiment with CHStone programs within LegUp HLS. There were several reasons for this decision. First, the CHStone benchmarks and synthesis contraints have been heavily optimised within the LegUp tool. Hence, it is a very good baseline, which may not necessarily produce a positive result on our method. It was important for us to showcase that our method may not be effective in some cases. Second, it is a considerable amount of time to efficiently port CHStone to Vitis HLS including the right optimisations and synthesis directives to achieve a strong enough baseline for experimentation.

Due to these limiting factors, we only explore one vertical arrow in Figure 4 from *PreciseCoarseLegUp* to *PreciseFineLegUp*. Please note that these factors only apply specifically to the CHStone experiments. Instead, we have add an extra dimension of discussion to CHStone: function inlining. The effects of fine-grained memory connections with and without function inlining shows how

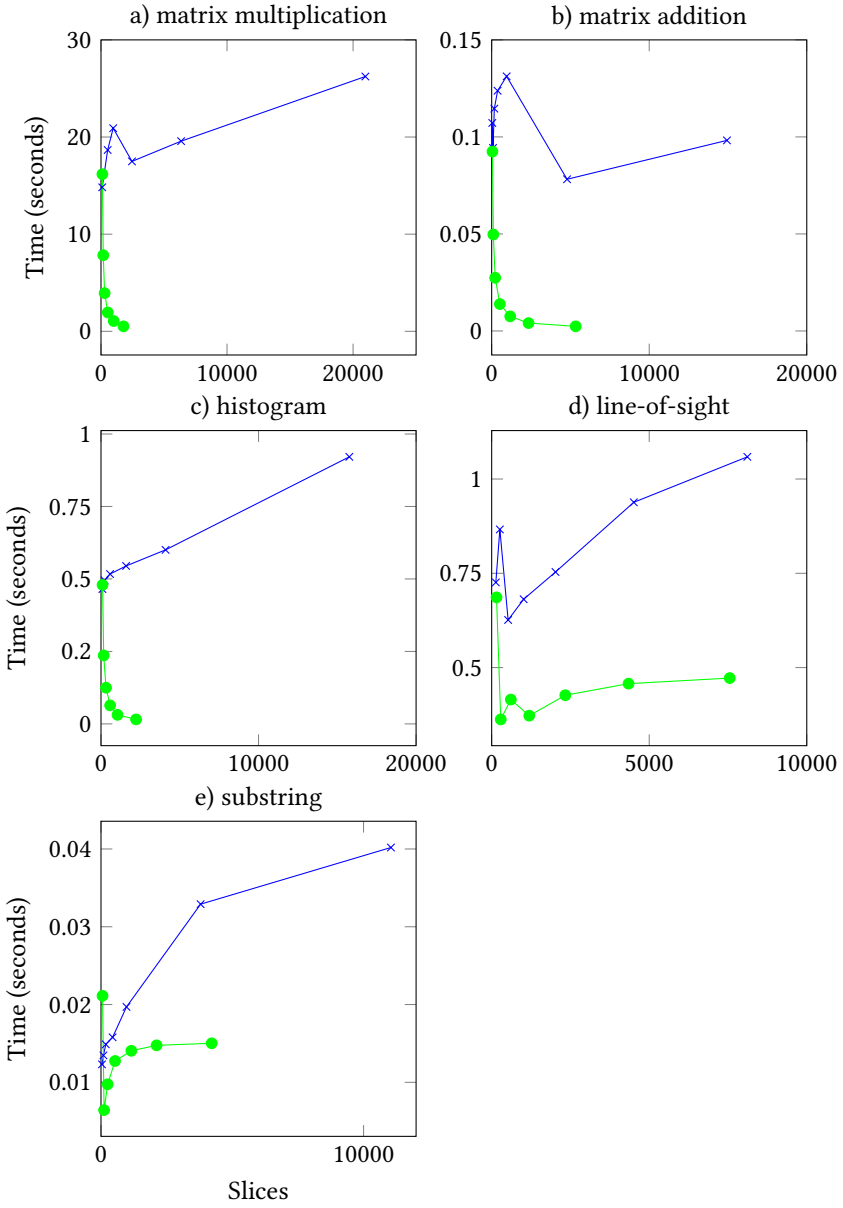


Fig. 10. Latency against area utilisation in Vitis HLS for both imprecise and precise analyses on HLS benchmarks exploring array partitioning by Cheng *et al.* [6]. We show imprecise fine-grained Vitis (*ImpreciseFineVitis*, —×—) , and precise fine-grained Vitis (*PreciseFineVitis*, —●—).

overoptimised HLS benchmarks can be and how the slightest change can offer new problems and insights.

Employing fine-grained connections in LegUp HLS for CHStone. However, introducing fine-grained memory connections can improve latency and area. Fig. 11(a) shows the summary of the relative

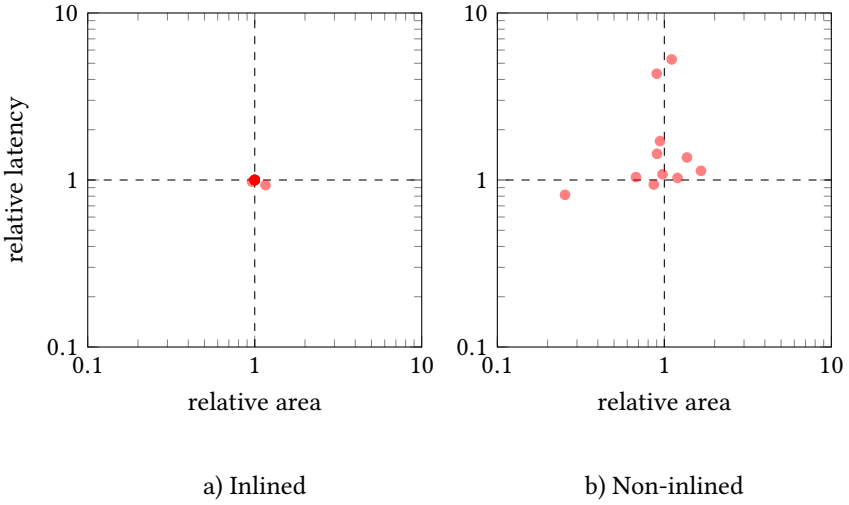


Fig. 11. Relative latency vs area of fine-grained LegUp (*ImpreciseFineLegUp*) of ChStone benchmarks (when enabling and disabling inlining), compared to original inlined LegUp HLS (*ImpreciseCoarseLegUp*).

latency vs area when implementing fine-grained connections. First, note that none of the data points are in the top half of the graph. This means our implementation never generates designs that are slower than the original CHStone synthesis in LegUp. Most programs lie at (1,1), which means they are not affected by fine-grained connections. This is because most CHStone programs have a points-to ratio (edge count divided by number of instructions) of one, that is $MultiAddrInst = \emptyset$. Only two programs have points-to ratio larger than one: *adpcm* and *jpeg*. These two programs, which are represented by the two points not at (1,1) in Fig. 11(a), run 6% and 3% faster with fine-grained connections. Fine-grained connections reduce area by 4% for *jpeg* but increase area by 15% for *adcpm*. This could be because *adpcm*'s points-to ratio of 1.5 is much larger than *jpeg*'s points-to ratio of 1.1.

The effects of fine-grained connections when function inlining is disabled. The points-to ratio of CHStone programs are optimised for inlining. When we disable inlining, the points-to ratio of most CHStone programs become larger than one. In general, inlining small trivial functions enables localised compiler optimisations, thereby reducing latency. However, inlining large complex functions leads to replication of large functions, thereby increasing area. Fig. 11(b) shows the relative latency vs area when implementing fine-grained connections (*ImpreciseFineLegUp*) after disabling inlining for CHStone programs, compared to the inlined CHStone within original LegUp (*ImpreciseCoarseLegUp*).

We see that most programs are slower, since most points reside in the top half of the graph. On average, *ImpreciseFineLegUp* is 60% slower than *ImpreciseCoarseLegUp*, with a maximum of 5×. However, we also see that 8 out of 12 programs are on the left half of the graph. This suggests that *ImpreciseFineLegUp* can reduce area, compared to *ImpreciseCoarseLegUp*. On average, circuits generated by *ImpreciseFineLegUp* are 10% smaller than *ImpreciseCoarseLegUp*. Circuits generated by *ImpreciseFineLegUp* can also be up to 65% larger than *ImpreciseCoarseLegUp*. We also see a corner case where when disable inlining, *ImpreciseFineLegUp*'s implementation of *motion* is 75% smaller and 20% faster than *ImpreciseCoarseLegUp*, simply by disabling inlining since the points-to ratio remained one and unchanged.

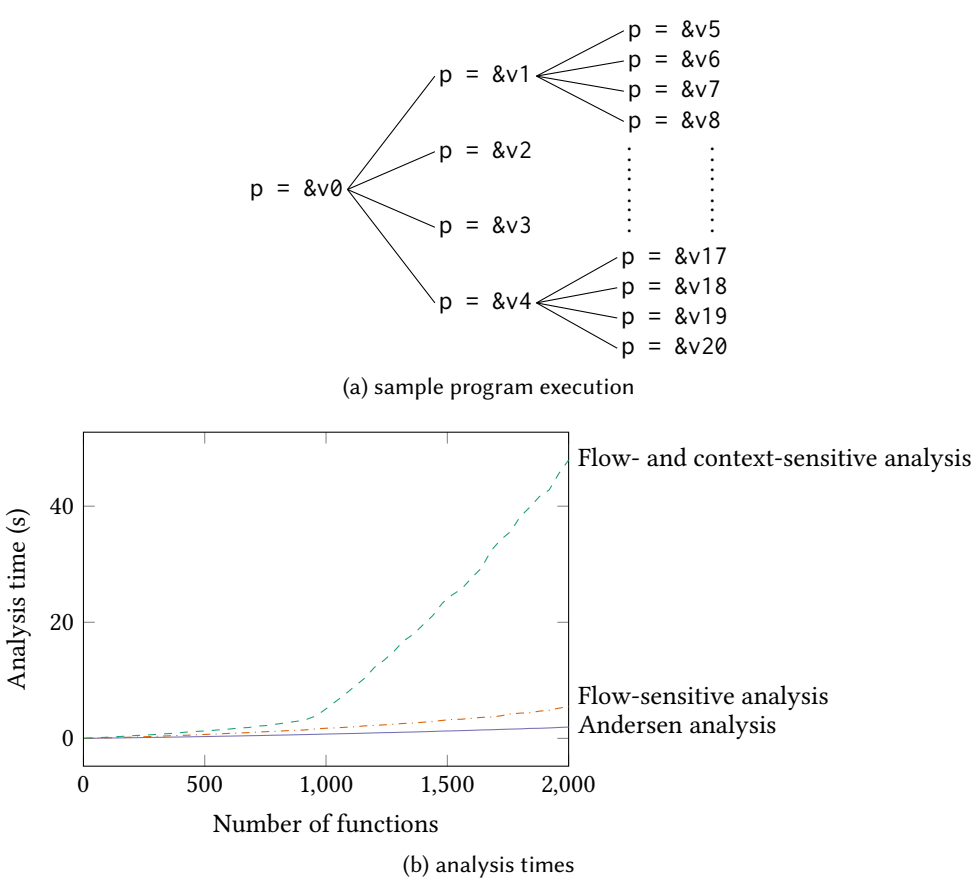


Fig. 12. An example where the input program causes analysis times of precise analysis to grow significantly.

4.4 Worst-case analysis

There are programs that can cause the time taken for precise analysis to grow significantly. A natural way to construct programs that are challenging to analyse precisely is to use recursion. However, recursion is generally unsupported by HLS tools. So, we devise a pathological program consisting of numerous non-recursive function calls, each assigning different addresses to a pointer. Fig. 12a shows a sample program execution with three levels of call depth, where each node is a function call. Fig. 12b shows the analysis times of all three analyses, as we scale the call depth of this synthetic program. We see that analysis time of flow- and context-sensitive analysis grows significantly faster than the other two analyses. This example suggests that the time penalty of using precise analysis is only noticeable for programs that are rather large or contrived, both of which are unlikely to appear in practice.

4.5 Summary of results

In this section, we have evaluated our implementations on three benchmark suites with different properties. The PTABen benchmark suite is a set of programs that use pointers in non-trivial ways. Our implementations generate PTABen hardware that reduces area and latency by 67% and 49% in

LegUp HLS (40% and 26% in Vitis HLS). These results provide a good reflection of how future HLS tools should adapt to growing program complexities in the software world.

The array partitioning benchmark suite is a set of programs that use arrays extensively and include various HLS angles such as memory partitioning, aliasing and loop pipelining. Our implementations generate array-partitioned hardware that reduces area and latency by 11% and 16% in LegUp HLS (94% and 92% in Vitis HLS).

Finally, the CHStone benchmark suite is a standard HLS suite that represents the different application domains synthesised via HLS. Our implementations generate CHStone hardware that reduces latency by 2% on average in LegUp HLS, and is never slower than the original implementation. We also show that CHStone programs are heavily optimised for inlining. When we disable inlining, our implementations can reduce area by 10% but increase latency by 60%.

5 RELATED WORK

In this section, we discuss existing pointer synthesis within HLS and the different existing pointer analyses in software.

5.1 Existing pointer synthesis in HLS

Séméria *et al.* [33] were the first to support the synthesis of pointers via HLS. Their method first replaces loads and stores to pointers with case statements and then encodes the addresses of each case label. Our work is inspired by theirs in that we also attempt to enumerate possible loads and stores before selecting the right choice. However, they never compared their method to any other architectures, most likely because they pioneered pointer synthesis. In addition, their implementation is based on the SUIF compiler framework, which is outdated and hardly used. Furthermore, they only tested very few programs that were synthetically generated and one CHStone program: *jpeg*. These reasons made it hard for us to compare against their work in a meaningful manner. In this article, we intend to perform these comparisons systematically, especially in the context of improving points-to precision. Our article is the first to evaluate several implementations across multiple benchmarks and tools.

Pilato *et al.* [28] propose implement memory accesses within high-level synthesis in a semi-automatic framework. Their approach is of addressing pointers is coarse-grained. They implement pointers as input ports, requiring runtime addressing for pointer disambiguation. They also propose global interconnections that connects all module to each other using elaborate memory interfacing, in a ring-like fashion [28, Figure 7]. Additionally, all memories are targeted as BRAM regardless of whether they are variables or arrays. Finally, they require users to use `pragma`s to allocate memories either internally or externally to a function. Their methodology of supporting pointers is very different to LegUp or Vitis HLS, since pointers are allocated directly in DRAM, which is possibly off-chip. As such, it was not possible to compare their work to ours.

Two prior works, by Séméria *et al.* [33] and Zhu *et al.* [51], support precise pointer analysis within HLS. However, neither work empirically studies the effects of precision on hardware synthesis. Séméria *et al.* developed their tool within the SUIF framework [42], which has a flow- and context-sensitive pointer analysis developed by Wilson and Lam [43].

5.2 Existing pointer analysis in software

Insensitive pointer analyses. There are several insensitive pointer analyses [2, 4, 38]. Two well-known insensitive pointer analyses are Andersen [2] and Steensgaard analysis [38]. Both these approaches can be viewed as constructing a constraint graph with variables as nodes and points-to relations as edges and then computing the transitive closure of this graph.

Among all the insensitive pointer analyses, Andersen analysis is the most-widely used algorithm [17, 36]. Hence, there have been many attempts to improve the analysis times of Andersen analysis including Hardekopf *et al.* [13]. Hardekopf *et al.* noticed that cycles between nodes result in the same points-to set for all those nodes and hence proposed online cycle detection to improve the practical running time of Andersen analysis. LegUp HLS uses an LLVM version of this algorithm [21, §4.11].

Precise pointer analyses. There are several flow-sensitive analyses. Most flow-sensitive analyses are implemented as data-flow analysis [7, 15], that iteratively propagates information across the entire program. Although this propagation improves precision, it necessitates significantly more compute time and memory. Hence, more recent tools [14, 39] utilise sparsity to reduce time and memory complexity.

There are several context-sensitive pointer analyses. Some of these analyses are flow-insensitive [20, 23, 26, 41]. Several other analyses are both flow- and context-sensitive [9, 37, 40, 43, 51]. Although most these tools would return the same points-to graph, several options were either built using non-LLVM compiler framework [9, 43], written in Java [37] or based on symbolic analysis [51], thus requiring redesign. Hence, in this paper, we utilise SUPA [40], an extension of SVF [39] built in LLVM, which supports flow- and context-sensitive pointer analysis.

6 CONCLUSION

In this article, we implemented methods to improve the precision and connection granularity of pointer synthesis within HLS tools. Our implementations overcome practical HLS inefficiencies that focus on the generation and interpretation of the points-to graph. Additionally, our article is the first to systematically and comprehensively test these different implementations across several benchmarks and HLS tools.

Firstly, we have shown that implementing a more precise analysis (flow-and-context-sensitive) reduces unnecessary sharing of HLS memory resources, with minimal impact on analysis times. Secondly, we have shown that introducing fine-grained memory connections further improve the sensitivity of the hardware connections to the points-to precision. Fine-grained memory connections introduces on per-instruction basis reduce unnecessary sharing of memory connections and aliasing during memory scheduling. We show that when both precision and connection granularity are improved, we can reduce area and latency by around 42% and 37% across three benchmark suites, ranging from non-trivial pointer use to standard HLS benchmarks, and two HLS tools.

As the complexity of pointer-based programs synthesisable via HLS increases, pointer synthesis will become increasingly important. We hope this work motivates research into how HLS tools compute the points-to graph and how back-end generators analyse, customise and implement this graph in hardware. We also hope this work motivates research into considering more complex memory elements in the points-to graph such as heaplets, stacks (function scoping) and C structures required to support data structures like linked lists and trees. Another interesting direction that we did not consider in this work is pointer use within multi-threaded programs.

REFERENCES

[1] [n.d.]. ([n. d.]). Supplementary material on GitHub, https://github.com/nadeshr/ppa-hls.
[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.
[3] Berkeley Design Technology, Inc. 2010. *An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool.* Technical Report. http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf.
[4] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. 1994. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer,

234–250.

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Field-Programmable Gate Arrays (FPGA)*.

[6] Jianyi Cheng, Shane T Fleming, Yu Ting Chen, Jason H Anderson, and George A Constantinides. 2019. EASY: Efficient Arbiter SYnthesis from Multi-threaded Code. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 142–151.

[7] Jong-Deok Choi, Michael Burke, and Paul Carini. 1993. 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*. 232–245.

[8] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. 2009. An Introduction to High-Level Synthesis. *IEEE Design and Test of Computers* 26, 4 (2009).

[9] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices* 29, 6 (1994), 242–256.

[10] Nicholas V Giamblanco and Jason H Anderson. 2019. ASAP: Automatic Sizing and Partitioning for Dynamic Memory Heaps in High-Level Synthesis. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 275–278.

[11] Nicholas V Giamblanco and Jason H Anderson. 2019. A Dynamic Memory Allocation Library for High-Level Synthesis. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 314–320.

[12] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 1192–1195.

[13] Ben Hardekopf and Calvin Lin. 2007. 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*. 290–299.

[14] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. *ACM SIGPLAN Notices* 44, 1 (2009), 226–238.

[15] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 289–298.

[16] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. *ACM SIGPLAN Notices* 36, 5 (2001), 24–34.

[17] Michael Hind and Anthony Pioli. 2000. Which pointer analysis should I use?. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. 113–123.

[18] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 898–911.

[19] Chris Lattner and Vikram Adve. 2002. The LLVM instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS* (2002).

[20] Chris Lattner and Vikram Adve. 2003. *Data structure analysis: An efficient context-sensitive heap analysis*. Technical Report. Tech. Report UIUCDCSR-2003-2340, Computer Science Dept., Univ. of Illinois ….

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

[22] LegUp Computing Inc. 2017. LegUp 6.4 Documentation. (2017). https://bit.ly/legup-memory-controller.

[23] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and scalable context-sensitive pointer analysis via value flow graph. *ACM SIGPLAN Notices* 48, 11 (2013), 85–96.

[24] Tingyuan Liang, Jieru Zhao, Liang Feng, Sharad Sinha, and Wei Zhang. 2018. HI-DMM: High-performance dynamic memory management in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2555–2566.

[25] Grant Martin and Gary Smith. 2009. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers* 26, 4 (2009), 18–25.

[26] Erik M Nystrom, Hong-Seok Kim, and W Hwu Wen-mei. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *International Static Analysis Symposium*. Springer, 165–180.

[27] Christian Pilato and Fabrizio Ferrandi. 2013. 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, 1–4.

[28] Christian Pilato, Fabrizio Ferrandi, and Donatella Sciuto. 2011. A design methodology to implement memory accesses in high-level synthesis. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 49–58.

[29] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.
[30] Nadesh Ramanathan, George A Constantinides, and John Wickerson. 2018. Concurrency-Aware Thread Scheduling for High-Level Synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 101–108.
[31] Nadesh Ramanathan, George A Constantinides, and John Wickerson. 2020. Precise Pointer Analysis in High-Level Synthesis. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 220–224.
[32] Nadesh Ramanathan, John Wickerson, and George A Constantinides. 2017. Scheduling Weakly Consistent C Concurrency for Reconfigurable Hardware. *IEEE Trans. Comput.* 67, 7 (2017), 992–1006.
[33] Luc Séméria and Giovanni De Micheli. 1998. 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*. 340–346.
[34] Luc Séméria and Giovanni De Micheli. 2001. Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 2 (2001), 213–233.
[35] Luc Séméria, Koichi Sato, and Giovanni De Micheli. 2001. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9, 6 (2001), 743–756.
[36] Yannis Smaragdakis, George Balatsouras, et al. 2015. Pointer analysis. *Foundations and Trends® in Programming Languages* 2, 1 (2015), 1–69.
[37] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. 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.
[38] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.
[39] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.
[40] Yulei Sui and Jingling Xue. 2018. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Transactions on Software Engineering* (2018).
[41] John Whaley and Monica Lam. 2007. *Context-sensitive pointer analysis using binary decision diagrams*. Ph.D. Dissertation. Citeseer.
[42] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices* 29, 12 (1994), 31–37.
[43] Robert P Wilson and Monica S Lam. 1995. Efficient context-sensitive pointer analysis for C programs. *ACM Sigplan Notices* 30, 6 (1995), 1–12.
[44] Felix Winterstein. 2017. *Separation Logic for High-level Synthesis*. Springer.
[45] Felix Winterstein, Samuel Bayliss, and George A Constantinides. 2013. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 362–365.
[46] Xilinx. 2018. *Vivado Design Suite User Guide: High-Level Synthesis (v2018.2)*.
[47] Xilinx. 2020. *Vitis Unified Software Platform Documentation: Application Acceleration Development (v2020.1)*.
[48] Zeping Xue and David B Thomas. 2015. SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.
[49] Zeping Xue and David B Thomas. 2016. SynADT: Dynamic data structures in high level synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 64–71.
[50] Y. Sui and J. Xue. 2020. PTABen Benchmark suite. (2020). https://github.com/SVF-tools/PTABen.
[51] Jianwen Zhu. 2005. Towards scalable flow and context sensitive pointer analysis. In *Proceedings. 42nd Design Automation Conference, 2005*. IEEE, 831–836.
[52] Jianwen Zhu and Silvian Calman. 2004. Symbolic pointer analysis revisited. In *PLDI*. ACM, 145–157.