Synthesizing Hot Code Paths by Abductive Reasoning

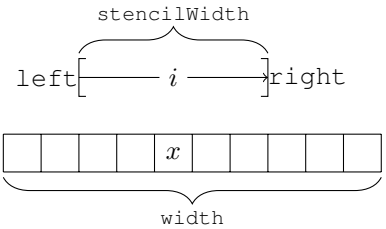
Simon Moll
Saarland University
moll@cs.uni-saarland.de

Sebastian Hack
Saarland University
hack@cs.uni-saarland.de

***Abstract*—Many SPMD programs suffer from divergence in control flow and memory accesses, e.g. when handling boundary conditions. While often only few work items diverge, a vectorizing compiler has to generate code that handles all of them. In general, this leads to overhead even for work groups that show no divergence. We present a novel optimization that synthesizes conditionals using abductive reasoning and Counter-Example Guided Inductive Synthesis (CEGIS) to dispatch diverging and non-diverging work items to specialized code paths. We evaluate our technique with a vectorizing OpenCL driver. On several benchmarks that the driver successfully vectorizes, we observe an additional speed up of up to 54.55% for control-flow optimizations and up to 146.14% for memory-access optimizations.**

I. Introduction

Many SPMD programs suffer from divergence in memory accesses and control-flow. Memory access divergence means that the addresses to load from are neither identical for all work items in a work group nor consecutive. Control-flow divergence means that the work items of a work group take different directions at a branch. Consider, for example, the one-dimensional convolution kernel depicted in Figure 1. Convolution is representative for operators in the image processing domain, where often the same kernel function is applied to every pixel individually.

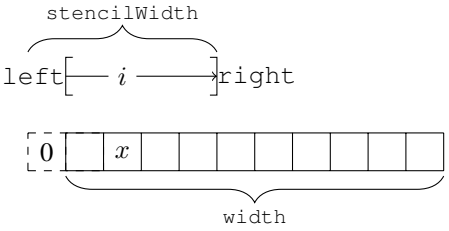



```
vstep = (stencilWidth - 1) / 2
for (x = 0; x < width; ++x) {
  left    = max(0, x - vstep)
  right   = min(width - 1, x + vstep)
  sum = 0

  for(i = left; i <= right; ++i) {
    sum = sum + input[i] * stencil[i - (x - vstep)]
  }
  output[x] = sum
}
```

Fig. 1: One dimensional convolution kernel.

Assume that a vectorizing compiler vectorizes the outer loop, which iterates over all pixels. The inner loop, which iterates over the stencil elements, executes sequentially for each work item. Let us assume that pixels beyond the image boundary do not contribute to the result (in practice, other techniques are also relevant). The stencil loop is optimized to only iterate over stencil elements that are multiplied with pixels in the image. This makes the trip count of stencil loop dependent on the pixel position.



(a) Instance of the convolution kernel with $x = 1$. The bounds of the inner stencil loop are optimized in a way that makes its trip count dependent to the pixel position x.

trip \ x	0	1	2	3	4	5	6	7	..
0	0	0	0	1	2	3	4	5	
1	1	1	1	2	3	4	5	6	
2	2	2	2	3	4	5	6	7	..
3		3	3	4	5	6	7	8	
4			4	5	6	7	8	9	..

(b) Table showing the mapping of pixel position (column) and loop trip (row) to value of `i`. The stencil has width 5. The first SIMD group (pixels 0 to 3) shows divergence in the loop trip count and values of `i`. The next SIMD group (pixels 4 to 7) is non-divergent with an uniform loop trip count and a consecutive pattern in the values of `i`. The highlighted column corresponds to the instance shown in Figure 2a.

Fig. 2: Divergent behavior in the convolution kernel.

As shown in Figure 2, this leads to control flow divergence at the image boundary because the trip count of the inner loop differs among work items of the same group. Vectorizing compilers [1] usually handle divergence by predication. Figure 3 shows the code such a compiler would have produced. Notice the predicated loads at the end of the inner loop that handle control flow divergence.

Work groups that only process pixels in the image center however, have non-divergent control flow and the value of `i`

```
vstep = (stencilWidth  - 1) / 2

// vectorized loop
for (x = 0; x + 3 < width; x = x + 4) {
  [x1,x2,x3,x4] = x + [0,1,2,3]
  left  = max(0, [x1,x2,x3,x4] - vstep)
  right = min(width - 1, [x1,x2,x3,x4] + vstep)
  sum   = [0,0,0,0]

  // stencil loop
  i = left
  do {
    [p1,p2,p3,p4] = i <= right
    if (!p1 && !p2 && !p3 && !p4)
      break

    [a1,a2,a3,a4] = i - (x - vstep)
    [in1,in2,in3,in4] = [0,0,0,0]
    [s1,s2,s3,s4] = [0,0,0,0]

    if(p1) { in1=input[i1]; s1=stencil[a1] }
    if(p2) { in2=input[i2]; s2=stencil[a2] }
    if(p3) { in3=input[i3]; s3=stencil[a3] }
    if(p4) { in4=input[i4]; s4=stencil[a4] }
    v = [in1,in2,in3,in4] * [s1,s2,s3,s4]
    sum = sum + v
    i = i + 1
  } while(true)
  output[x:x + 3] = sum
}
```

Fig. 3: With predicating vectorization. The compiler vectorizes the outer loop that iterates over the pixel coordinates `x`. As its trip count varies by the pixel, the inner loop is fully predicated: the loop iterates until all work items have finished (`p1` to `p4`).

```
vstep = (stencilWidth  - 1) / 2
sum   = [0,0,0,0]
left  = x - vstep
right = x + vstep

for(i = left; i <= right; ++i) {
  pixel = input[i:i + 3]
  weight = stencil[i - (x - vstep)]
  sum = sum + pixel * weight
}

output[x:x + 3] = sum
```

Fig. 4: Vectorized loop body for SIMD groups in the image center.

is consecutive in the pixel position. For all of the work items in such a work group, either `p1`–`p4` are either all true or all false. For these work items, the more efficient code show in Figure 4 could be used. This code does not need to handle the divergence imposed by the boundary condition. The counter variable `i` is *consecutive* and increases by one from one instance to the next. A single vector load instruction accesses the elements of the `input` array indexed by `i`. Similarly, the load to the `stencil` array is uniform and remains scalar. No predication is needed.

The condition under which the uniform loop can be executed is checked quickly by the following test and without any vector instructions:

$$(\mathtt{x} \geq \mathtt{vstep}) \wedge (\mathtt{x} + \mathtt{vstep} + 3 < \mathtt{width}) \tag{1}$$

This condition is independent of the i loop iterating over the stencil elements. Thus, the compiler can optimize the x loop by splitting it at the boundaries of the optimization condition. This generates a code path for the left image margin, the optimized code path for center pixels and again a path for the right image margin as shown in Figure 5. The condition is simple enough such that the compiler achieves this transformation this with standard optimizations such as loop splitting and unswitching. However, to the best of our knowledge no existing compiler optimization is able to infer this condition automatically.

In this paper, we present a technique that is able to infer equation 1 *automatically* by employing a mode of reasoning called *abduction*. Given a proposition N, one looks for a possible explanation M, such that $M \implies N$. The proposition M is then said to be abducted form N. We use abduction to infer from constraints, under which more aggressive program optimization would be possible, a condition that guards a path where the optimizations are applicable.

Coming back to our example, our technique starts with the vectorized program (Figure 3). We want to find a condition under which the control flow of the inner stencil loop is preserved. To this end, we state the initial proposition (2) that the exit condition of the loop is *uniform*.

$$(\mathtt{i} <= \mathtt{right}) : \mathtt{U} \tag{2}$$

$$\Longleftarrow \mathtt{left} : \mathtt{C} \wedge \mathtt{right} : \mathtt{C} \tag{3}$$

$$\Longleftarrow (\mathtt{x} \geq \mathtt{vstep}) \wedge (\mathtt{x} + 3 < \mathtt{width} - \mathtt{vstep}) \tag{4}$$

Our procedure implements two kinds of abductive reasoning. In *Abstract Abduction*, the procedure assigns more precise abstractions to program variables such that the proposition holds. We apply abstract abduction whenever the proposition uses loop-carried variables as it is the case in our example. The procedure abducts proposition (15), which states that `left` and `right` are consecutive. This condition could be tested in the outer loop at run time. However, we generate a more efficient test, that can be tested outside of both loops, through a second kind of abduction. *Concrete Abduction* synthesizes a predicate on the program variables that implies the proposition. This finally leads to proposition (4), which directly checks if the SIMD group is in the image center.

A. Contributions

In this paper, we make the following contributions.

- We present a novel synthesis procedure for boolean predicates over linear integer terms. The procedure combines SMT solving with edge detection in binary images to implement a variant of the Counter-Example Guided Inductive Synthesis (CEGIS) paradigm.
- We apply our synthesis procedure in the context of whole-function vectorization [1]. The procedure synthesizes efficient tests for code paths that exploit optimizable memory and control flow patterns. To our knowledge, these optimizations were before only possible with speculative approaches or profiling data. Our technique is

```
int vstep = (stencilWidth  - 1) / 2
int x = 0;
for(;x - vstep < 0; x +=4) {
  ... //default
}

for (; x + 3 + vstep < width; x += 4) {
  float4 sum = 0
  int left   = x - vstep
  int right  = x + vstep

  for(int i = left; i <= right; ++i) {
    float4 pixel = input[i:i + 3]
    float weight = stencil[i - (x - vstep)]
    sum += pixel * weight
  }
  output[y * width + x] = sum
}

for(;x + 3 < width; x +=4) {
  ... // default
}
```

Fig. 5: With optimized path for center pixels. The pre-existing loop splitting pass in LLVM optimizes the program further.

applicable at compile time. Although evaluated in this specific domain, our technique is generally applicable.

- We evaluated our technique experimentally on a research OpenCL driver, which uses a state-of-the-art Whole-Function Vectorizer based on LLVM. We observe an additional speed-up of up to 146.14% over functions vectorized without the optimization. The experiments also confirm the efficiency of the synthesis procedure in terms of compile time: It reliably terminates within seconds.

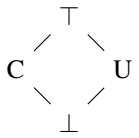
This paper is organized as follows. In Section II, we give an overview of the steps involved in our optimization. We continue in Section III with a detailed explanation of, first, the optimization algorithm and, second, the actual synthesis procedure. We report experimental results in Section V and conclude in Section VI.

II. Overview

We give a schematic overview of our optimization procedure in Figure 6. The vectorizing compiler invokes the optimization procedure on the scalar input function and provides a constraint that describes the hot code path to optimize. In our example, this is the condition $\alpha(i < \mathit{right}) \sqsubseteq U$. In this section, we outline the steps involved in the optimization starting from the moment of invocation.

A. Abstract Abduction.

If the hot path contains loop-carried variables, as it is the case in our example, we need a loop invariant to reason about the program and the condition statically. In this work, we use the invariants provided by divergence analysis [2], [1]: Current vectorizers use data flow analysis to restrict the locations where divergence can occur. If all operands of an instruction are identical for all vector elements, we say that the operation is *uniform* and the corresponding instruction may remain scalar in the vectorized program. The analysis classifies control-flow masks and program variables according to a divergence lattice. We show the parts of the lattice relevant to our work in Figure 7. Note, that control-flow masks, which implement predication in the vector program are boolean and are therefore abstracted to either *uniform* or $\top$. Figure 8 shows the results



Abstract	Concrete
$\top$	$[x, y, z, w]$
C	$[n, n+1, n+2, n+3]$
U	$[n, n, n, n]$

Fig. 7: Hasse diagram of the simplified divergence lattice $\mathbb{L}$ without alignment (for vector width 4). The *abstraction function* $\alpha : \mathbb{Z}^4 \to \mathbb{L}$ maps concrete valuations to lattice elements.

of divergence analysis on our running example. The divergent behavior of the stencil loop reflects in the $\top$ annotations for the variables `i`,`left` and `right` controlling it.

```
stencilWidth:U
vstep:U = (stencilWidth - 1) / 2
for (x = 0; x < width; ++x) {
  x:C
  left:⊤  = max(0, x - vstep)
  right:⊤ = min(width - 1, x + vstep)
  sum:U = 0

  for(i:⊤ = left:⊤; (i <= right):⊤; ++i:⊤) {
    sum = sum:⊤ + input[i]:⊤
                * stencil[(i - (x - vstep)):⊤]:⊤
  }
  output[x:C] = sum:⊤
}
```

Fig. 8: One dimensional convolution kernel as it is annotated by the divergence analysis. Here, `v:l` denotes that variable `v` receives the abstraction `l` with respect to the divergence lattice of Figure 7.

The condition under which we can make our optimization is that

$$Z := \alpha(i \leq \mathit{right}) \sqsubseteq U$$

where α is the abstraction function (in the sense of abstract interpretation) of divergence analysis that maps concrete to abstract values. Condition Z means that the comparison $i \leq \mathit{right}$ yields either true or false *for all* vector elements, that is the loop has no control flow divergence. Now we want to abduct a condition X that implies Z but comprises only variables that are *not* carried by the loop. Therefore, X can be checked before the loop. In abstract abduction, this condition is expressed on the abstract values of a divergence analysis.

To this end, we cast the divergence analysis of the input program as an SMT formula. This formula encodes the data flow equations and the abstract semantics that relates abstract to concrete states. The divergence analysis of the loop-carried variable i in our running example is represented by the

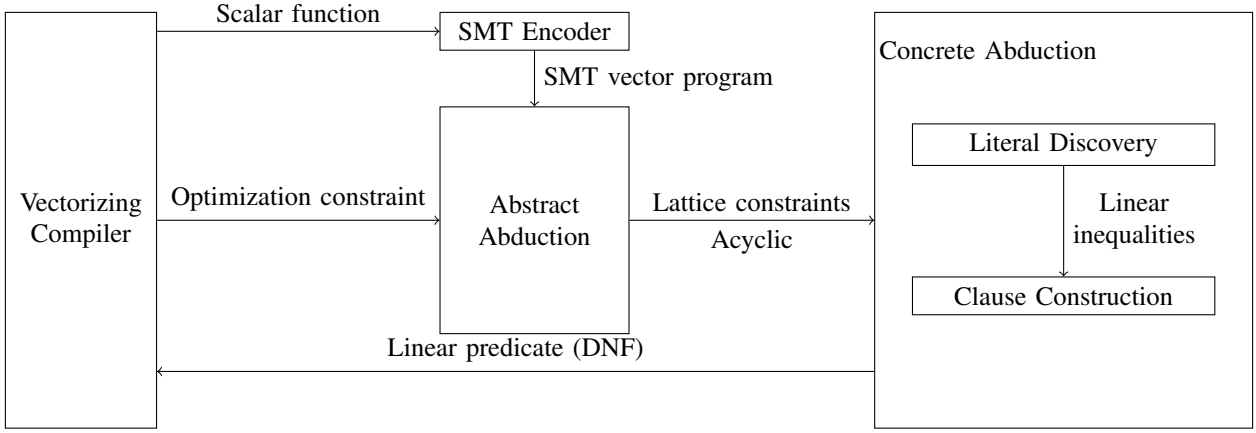


Fig. 6: Schematic overview of the optimization procedure.

following formula:

$$\begin{aligned}
&\phi := \\
&\exists \mathit{left}^\sharp, \mathit{right}^\sharp, \mathit{ipp}^\sharp, i^\sharp. \\
&\quad \Big(i^\sharp \sqsubseteq \mathit{left}^\sharp \sqcup \mathit{ipp}^\sharp \wedge \\
&\forall \sigma. \\
&\quad \big(\alpha(i(\sigma)) \sqsubseteq i^\sharp \implies \alpha(\mathit{ipp}(\sigma)) \sqsubseteq \mathit{ipp}^\sharp\big) \wedge \\
&\quad \alpha(\mathit{right}(\sigma)) \sqsubseteq \mathit{right}^\sharp \wedge \\
&\quad \alpha(\mathit{left}(\sigma)) \sqsubseteq \mathit{left}^\sharp \Big)
\end{aligned} \tag{5}$$

Any fixed point solution of formula (5) is a solution to the divergence analysis problem. The least fixed point solution with respect to $\mathit{left}^\sharp$,$\mathit{right}^\sharp$,$\mathit{ipp}^\sharp$ and $i^\sharp$ is the most precise solution. For example, a trivial fixed point is given by abstracting all variables to $\top$.

We find the least fixed point solution using a refinement loop based on program state samples σ. We start by setting all abstract variables $\mathit{left}^\sharp$,$\mathit{right}^\sharp$,$\mathit{ipp}^\sharp$ and $i^\sharp$ to $\bot$. Then, we repeatedly generate program states σ that are counter examples for the universal quantifier. If a counter example exists, we generalize the abstract variables such that the formula is again satisfied. The process terminates in a fixed point solution, for which no counter example states σ exist. In the case of the divergence lattice, this is the least fixed point solution. For brevity we will only the show the part of the program state samples relevant to the solution process.

Initially, all abstract variables are $\bot$ and so a possible counter example is:

$$\sigma_i = [0,0,2,1] \wedge \sigma_{\mathit{left}} = [3,7,6,2] \wedge \sigma_{\mathit{right}} = [4,8,9,3]$$

All evaluations in state σ are $\top$ and violate the current model that sets all abstractions to $\bot$. Therefore, we generalize the abstract variables such that σ is not longer a counter example of formula (5) by setting:

$$\mathit{right}^\sharp = \top \wedge \mathit{left}^\sharp = \top \wedge i^\sharp = \top$$

Note, that we can not modify the abstract variable $\mathit{ipp}^\sharp$ as the counter example does not satisfy the precondition $\alpha(i(\sigma)) \sqsubseteq i^\sharp$. We run the refinement loop again and might get the state sample

$$\sigma_i = [5,7,8,9] \wedge \sigma_{\mathit{ipp}} = [6,8,9,10]$$

We finally refine $\mathit{ipp}^\sharp = \top$. As all abstract variables are now $\top$ any program state is admissible and no further counter examples exist. The process outlined above generates a least fixed point solution of the data flow problem for all states. However, the optimization we want to perform, that is to produce a special variant of the inner loop for all non-divergent work items, is not valid *for all* states but only for those states for which condition Z holds.

To abduct X from Z, we modify the analysis equation in the following way: We add the optimization constraint Z to the data flow equation part of ϕ. This restricts the set of states to those for which Z is holds. In general, this alone makes the formula unsatisfiable. Therefore, we relax the formula in the sense that we allow the solver to select a more restrictive abstraction to potentially satisfy the formula. In the example, we will relax the abstract variables $\mathit{right}^\sharp$ and $\mathit{left}^\sharp$

$$\begin{aligned}
&\phi := \\
&\exists \mathit{left}^\sharp, \mathit{right}^\sharp. \exists \mathit{ipp}^\sharp, i^\sharp. \\
&\quad \Big(i^\sharp \sqsubseteq \mathit{left}^\sharp \sqcup \mathit{ipp}^\sharp \wedge \\
&\forall \sigma. \\
&\quad \big(\alpha(i(\sigma)) \sqsubseteq i^\sharp \wedge \alpha(\mathit{right}(\sigma)) \sqsubseteq \mathit{right}^\sharp \wedge \alpha(\mathit{left}(\sigma)) \sqsubseteq \mathit{left}^\sharp \\
&\quad \implies \\
&\quad \alpha(\mathit{ipp}(\sigma)) \sqsubseteq \mathit{ipp}^\sharp \wedge \\
&\quad \alpha(i(\sigma) \leq \mathit{right}(\sigma)) \sqsubseteq U)\big)\Big)
\end{aligned}$$

In this abduction problem, we are interested in a greatest fixed point solution in the relaxed variables that is consistent with respect to Z. The restriction formula corresponding to a greatest fixed point solution that is Z-consistent corresponds is a weakest precondition for the optimization constraint.

This new problem, has the new trivial solution that sets all relaxed variables to $\bot$.

If we fix the relaxed variables to abstractions, we get again a divergence analysis problem. We use this observation to split the abduction problem in two parts, the inner data flow problem and the outer abduction problem on the relaxed variables. We solve the inner data flow problem with the method outlined above. For the outer abduction problem, we search for a greatest assignment to the relaxed variables such that the inner problem has a solution that is consistent with the optimization constraint Z. To this end, we initially set $\mathit{left}^\sharp$ and $\mathit{right}^\sharp$ to $\bot$. Each iteration generalizes the relaxed variables, solves the data flow problem and checks whether the solution satisfies the optimization constraint. If no further generalization is possible, the process terminates.

For our running example, this finds the following greatest assignment to the relaxed variables, which corresponds to the abducted formula

$$X := \alpha(\mathit{left}) \sqsubseteq C \wedge \alpha(\mathit{right}) \sqsubseteq C$$

B. Concrete Abduction.

Abstract abduction derived constraints expressed by means of the abstraction used in divergence analysis. Concrete abduction abducts a linear integer predicate that implies the constraint found by abstract induction. This is desirable as linear constraints on loop indices are easily exploitable by standard compiler optimizations as loop splitting (cf. Figure 5). Furthermore, the linear predicates we abduct do not use vector variables making them cheap to evaluate.

We devised a novel synthesis procedure that under-approximates a boolean predicate s over a vector of integers, $\Sigma \subseteq \mathbb{Z}^n$ The procedure synthesizes a predicate p in disjunctive normal form (DNF) where each literal has the form

$$\pm x \cdots \pm z \leq c \pmod m$$

Unlike other template-based synthesis procedures, the number of clauses, literals per clause, and integer variables per literal is *not limited a priori*.

The synthesis procedure operates in two phases. During *literal discovery* the method analyzes the specification s and proposes a set of literals L of above form. In our running example, the proposition (15) does hold at $(\mathtt{x}, \mathtt{vstep}, \mathtt{width}) = ([1,2,3,4], 1, 8)$. It is invalidated, if increasing `vstep` by 1 or if decreasing all components of `x` by 1. The proposition is tight in `vstep` and `x`, which is captured in the linear constraint $\sigma_x \geq \sigma_{\mathtt{vstep}}$. If the number of literals exceeds a given threshold, Literal Discovery proposes a modulus constraint that subsumes as many already found literals as possible.

In the final *boolean structure synthesis* phase, the procedure iteratively constructs conjunctive clauses that cover sections of s that evaluate to *true*. The clauses are build from the literals found during literal discovery.

At this point, we only state that the procedure will find the following solution and refer the reader to Section III-E for a detailed presentation of the synthesis procedure:

$$p(\sigma) = (\sigma_x \geq \sigma_{vstep}) \wedge (\sigma_x + 3 < \sigma_{width} - \sigma_{vstep})$$

III. Methodology

The optimization applies abductive reasoning to abduct from the optimization constraint a quickly-checkable predicate on program variables. If the abducted predicate holds, the optimization constraint holds as well, and execution can continue on a path that is optimized for the constraint. The optimization procedure applies abductive reasoning through two specialized abduction procedures - *Abstract Abduction* and *Concrete Abduction*.

If the optimization constraint is loop carried, Abstract Abduction abducts an intermediate constraint that is not. This intermediate constraint is a conjunction of lattice constraints of loop-carried variables. Finally, Concrete Abduction synthesizes a linear predicate in Presburger Arithmetic that implies the optimization constraint. The invoking compiler generates a specialized code path conditioned on this predicate. We describe the SMT formulation of the Vectorization Analysis in Section III-A. In this work, we consider *Presburger Arithmetic*, the theory of inequalities over linear integer terms, for which the problem is decidable. We formally describe Abstract Abduction in Section III-B and Concrete Abduction in Section III-C. We finally specify our Synthesis Procedure in Section III-E.

A. SMT Program Encoding

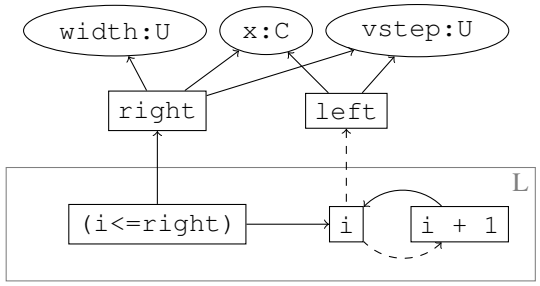


Fig. 9: SSA data dependency graph of the running example. Expressions in ellipses are fixed to the respective lattice abstraction. Dashed edges denote data dependency through abstract interpretation. Nodes in the gray-shaded rectangle are located in the inner-most loop of the convolution kernel.

The normal divergence analysis is a forward flow analysis with abstract semantics for the operators. In order to facilitate the backward reasoning required for abduction, we transfer the data dependency graph of the function into an SMT problem. We show the SSA data dependency graph of our motivating example in Figure 9.

The optimization constraint refers to properties of the vectorized function and thus the SMT encoding of the function needs to be vectorized. The SMT encoding step translates all variables of the function's scalar SSA data dependency graph into vectors in the SMT formulation. To this end, it

defines functions $v^{(l)}$ that compute the l-th component of vector variable v in program state σ.

- Function parameters and values from calls of unknown functions or memory accesses are represented as independent components of σ. The functions $v^{(l)}$ will only project their vector value from the program states. The projection functions may still encode prior knowledge about possible vector shapes. Lines (6) to (8) introduce the *uniform* variables $width$,$vstep$, and the *consecutive* variable x.
- Cyclic data dependencies are broken by representing the PHI nodes in loop headers as independent vector variables. The data dependencies of these variables are then approximated by means of abstract interpretation. The variable `i` is loop-carried and modeled as a vector variable in line (11). Constraint (12) preserves the data dependence to `left` and `i` $+1$ by abstract interpretation.
- We remove data dependencies through operations that exceed the capabilities of the SMT solver and make the dependent variables independent components of σ instead. For example, to stay in the realm of Presburger Arithmetic, we only allow division by a constant but not by a variable.

$$width^{(l)}(\sigma) = \sigma_{width} \tag{6}$$
$$vstep^{(l)}(\sigma) = \sigma_{vstep} \tag{7}$$
$$x^{(l)}(\sigma) = \sigma_x + l \tag{8}$$
$$left^{(l)}(\sigma) = max(0, x^{(l)} - vstep^{(l)}) \tag{9}$$
$$right^{(l)}(\sigma) = min(x^{(l)} + vstep^{(l)}, width^{(l)} - 1) \tag{10}$$
$$i^{(l)}(\sigma) = \sigma_i^{(l)} \tag{11}$$
$$\alpha(i(\sigma)) \sqsubseteq \alpha(left(\sigma)) \sqcup \alpha(i(\sigma) + 1) \tag{12}$$

Note, that we eliminate PHI nodes in acyclic data dependencies by if conversion, which represents the PHI as an expression selecting its value from all incoming values based on a boolean predicate. We will use v as a short hand for $v^{(l)}(\sigma)$, if not indicated otherwise.

B. Abstract Abduction

Abstract abduction restricts the abstractions of program variables such that the optimization constraint holds. Such a *restriction formula* has the form

$$\bigwedge_{v \in V_{rel}} \alpha(v) \sqsubseteq a_v$$

A program state σ satisfies a restriction formula, if its assignment σ_v to each variable v, is included in the restricting abstraction a_v. V_{rel} is the set of variables that may be restricted. If Abstract abduction succeeds it returns a general-most restriction formula. This means that the restriction on no variable can be weakened without admitting program states that violate the optimization constraint.

Abstract abduction is given a set of variables that it may restrict to make the data flow problem satisfiable under the optimization constraint. We call these variables *relaxed* as we may give them any valuation to make the problem satisfiable. We denote the set of relaxed variables as $V^\sharp_{rel}$. We choose as relaxed variables those variables that interact with the optimization constraint and that are immediate data dependences of loop carried variables. Consider the data flow problem of our running example shown in Figure 10. `right` and `left` are relaxed because they are used by the iteration variable `i`.

On the other hand, the data flow problem contains the *abstract* variable $i^\sharp$. This abstract variable is necessary to preserve the loop-carried data flow. $V^\sharp_{abs}$ is the set of abstract variables occurring in the data flow problem.

The abstract abduction formula has the general shape

$$\exists V^\sharp_{rel}. \left(\exists V^\sharp_{abs}. (\forall \sigma. (A(\sigma) \wedge M(\sigma) \implies N(\sigma))\right) \tag{13}$$

Formula (15) is existentially quantified in the relaxed and abstract variables. A is the restriction formula on the relaxed variables. The formulas M and N encode the data flow problem under the optimization constraint. M restricts concrete states σ to be consistent with the abstract variables in $V^\sharp_{abs}$. N contains the local consistency constraints of the data flow problem and the optimization constraint. We are looking for a greatest fixed point with respect to variables in $V^\sharp_{rel}$, such that the nested data flow problem is satisfiable.

Consider the abstract abduction problem of our running example shown in Figure 10.

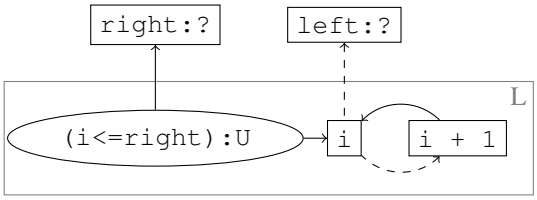


Fig. 10: SMT formulation of the abstract abduction step. A solution constrains the relaxed variables `right` and `left`, such that `i<=right` is *uniform*.

In this case, the formulas are given as

$$A := \alpha(right(\sigma)) \sqsubseteq right^\sharp \wedge \alpha(left(\sigma)) \sqsubseteq left^\sharp \tag{14}$$
$$M := \alpha(i) \sqsubseteq i^\sharp \tag{15}$$
$$N := \alpha(i+1) \sqsubseteq ipp^\sharp \wedge \alpha(i) \sqsubseteq ipp^\sharp \sqcup left^\sharp \tag{16}$$
$$\alpha(i \leq right) \sqsubseteq U \tag{17}$$

a) Problem partitioning: The partition of the abstraction variables into the two sets $V^\sharp_{rel}$ and $V^\sharp_{rel}$ divides the abstract abduction formula (15) into two nested problems. The parametric data flow problem

$$\phi(V^\sharp_{rel}) := \exists V^\sharp_{abs}. (\forall \sigma. (A(\sigma) \wedge M(\sigma) \implies N(\sigma))$$

for which, we are interested in a solution that is a least fixed point in $V^\sharp_{abs}$ and the abduction problem:

$$\exists V^\sharp_{rel}. \left(\phi(v^\sharp_{rel})\right)$$

for which we are looking for a greatest fixed point solution.

In the following, we will discuss how both problems can be solved independently using program state samples. We begin with the inner data flow problem.

1) Inner Data Flow Problem: For the inner data flow problem, the abstractions for relaxed variables are fixed and the problem reduces to a forward flow analysis. Therefore, we could use the abstract transformers of the default Divergence Analysis to find a least fixed point solution.

We use a refinement loop based on example states σ to find the least fixed point solution. We initially set $v^\sharp_{abs} := \bot$ and solve the following formula with an SMT solver

$$\exists\sigma.\ \bigvee_{v \in V^\sharp_{abs}} \alpha(\sigma_v) \sqcup v^\sharp_{abs} \sqsupset v^\sharp_{abs} \wedge (A(\sigma) \wedge M(\sigma) \implies N(\sigma)) \tag{18}$$

If formula (18) is unsatisfiable, there is no program state that is subsumed by the abstract variables $v^\sharp_{abs}$. Therefore, $v^\sharp_{abs}$ is the least fixed point solution. Otherwise, there is a program state σ, that is not covered by the abstract variables. In this case, we generalize the abstract variables such that they contain the sample σ by setting:

$$v^\sharp_{abs} := \alpha(\sigma_v) \sqcup v^\sharp_{abs}$$

After the data flow problem has been solved we check with an SMT solver, if there is a program state σ that is valid in the data flow problem but violates the optimization constraint. If so, we report to the outer solver procedure that no fixed point exists. Otherwise, the inner data flow problem is satisfiable and the assignment to the relaxed variables is consistent with the optimization constraint.

2) Outer Abduction Problem: A solution to the outer abduction problem is a greatest fixed point solution such that the inner data flow problem is satisfiable. We use fixed point iteration starting from a valuation that assigns $\bot$ to all relaxed variables.

The iteration function f has two arguments: the current restriction formula A and a validity formula C. While A is current restriction formula, C encodes constraints how the relaxed variables can be generalized. We set initially $C := \mathit{true}$ and $a_v := \bot$.

In each iteration, the function f queries for new restrictions a^+ that generalize the current restriction a in a way that is consistent with C.

$$\exists a^+_v \left(\bigvee_{v \in V^\sharp_{rel}} a^+_v \sqsupset a_v \right) \wedge C(a^+)$$

If the formula is unsatisfiable, the current restriction formula A is already a greatest fixed point solution and the algorithm terminates. If such a a^+ exists, we fix all relaxed variables and evaluate the resulting data flow problem. If the inner data flow problem has a fixed point solution that is consistent with the optimization constraint, we update the restriction formula by setting $a := a^+$ and repeat. Otherwise, a^+ is an invalid generalization and we update C to block it in future iterations by setting:

$$C'(a) := C(a) \wedge \left(\bigvee_{v \in V^\sharp_{rel}} \overline{a_v \sqsupseteq a^+_v} \right)$$

As C' is a restriction on C, the iteration formula descends on the lattice of validity functions. If the underlying lattice is finite, the process is bound to terminate with a valid restriction formula.

a) Motivation: Abstract Abduction is not strictly necessary and concrete abduction could be applied directly. If the optimization constraint is loop carried, concrete abduction will generate a loop-carried predicate. This means, there will only be a specialized code path for the loop body and the predicate needs to be re-checked in each iteration of the loop. On the other hand, if there is a non loop-carried implicant for a loop-carried optimization constraint, the implicant allows the creation of a path that specializes the loop completely. Abstract Abduction is also motivated by code quality - a non-$\top$ annotation for a program variable means that the compiler can generate more efficient code for it in the vectorized program. For our running example, Abstract Abduction will find the two implicants $\mathit{left} : C \wedge \mathit{right} : C$ and $\mathit{left} : U \wedge \mathit{right} : U$. By applying concrete abduction directly on the optimization constraint, these two solutions are lumped together. While the resulting linear predicate may apply for more program states, it will also imply that

$$\mathit{left} : \top \wedge \mathit{right} : \top$$

meaning the computation of left and right has to be done in vector registers. This is not the case, when applying concrete abduction to one of the interpolants found by abstract abduction.

b) Generalization & Limitations: Abstract abduction is applicable to other flow analysis, given that the underlying lattice is finite and the solver can handle the theory used in the abstraction function.

Note, that by modeling computation in the SMT domain on concrete valuations, our procedure does not need transfer functions. One may still use an abstract transfer function to improve the precision in the case where a new independent variable is introduced instead of computing its value from the program state. This would conserve some degree of data dependence.

C. Concrete Abduction

Concrete Abduction infers a linear integer predicate that implies a given condition. This is the hot path constraint, if the control flow in the vectorized loop is acyclic, or otherwise the condition found by Abstract Abduction. The SMT formula for Concrete Abduction is based on the SMT formulation of the Divergence Analysis problem. The condition is, in any case, fully computable from the program state and no $\sqcup$-operator is required in the construction of the formula. A solution to the concrete abduction problem is a predicate, $p : \Sigma \to \mathbb{B}$,

that selects a subset of the program states σ which makes the problem satisfiable:

$$\exists p \in \Sigma \to \mathbb{B}. \tag{19}$$

$$\forall \sigma \in \Sigma. (p(\sigma) \implies (M(\sigma) \implies N(\sigma))) \tag{20}$$

$$\exists \sigma^* \in \Sigma. (p(\sigma^*) \wedge M(\sigma^*)) \tag{21}$$

The formula has a second-order existential quantifier for the unknown predicate that will by synthesized. Predicates M and N are place holders for the data flow constraints introduced in Section III-A. Line (21) rules out the trivial solution $p := \mathit{false}$. This synthesis problem does not capture the notion that the predicate p should hold on as many program states σ as possible. We will present a formal grounding of this notion in the presentation of the synthesis procedure in Section III-E.

For our running example of Figure 1, the predicates M and N that encode the data flow constraints are:

$$M \models \sigma_{width} \geq 0 \wedge \sigma_{vstep} \geq 0 \wedge \sigma_x \geq 0 \tag{22}$$

$$N \models \alpha(min(\sigma_{width} - 1, \sigma_x + \sigma_{vstep})) \sqsubseteq C \tag{23}$$

$$\wedge\, \alpha(max(0, \sigma_x - \sigma_{vstep})) \sqsubseteq C \tag{24}$$

Variables ranges are encoded in line (22). Lines (23)-(24) encode the intermediate condition found by *Abstract Abduction*.

D. Consistency

It is not obvious, why the solution to the SMT-based Divergence Analysis should be consistent. This is because of the mixed use of abstract interpretation and computation, and the fact that some data dependencies are completely ignored in the SMT formulation. Consider the excerpt from a data dependency graph in Figure 11 annotated with lattice constraints found by abstract abduction. No actual program execution will satisfy the lattice constraints x/a:C and x/b:U in conjunction. When abducting a predicate, the only consequence of this is, that execution will never enter the synthesized path. We argue that, in the SMT Divergence Analysis step, ignoring all data dependencies of a variable and fixing its abstraction to top is a safe over approximation. This approximation yields consistent solutions at the cost of being imprecise.

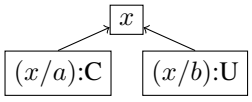


Fig. 11: Example of a hidden non-linear data dependency that will not transfer from the SSA data dependency graph to the SMT formulation. Instead x/a and x/b will be treated as independent variables of the program state. While the constraint (x/a):C and (x/b):U, generated by abstract abduction, is infeasible, this only means that the synthesized path will never be taken.

E. Synthesis Procedure

We devised a novel synthesis procedure that under-approximates a boolean predicate s over a vector of integers, $\Sigma \subseteq \mathbb{Z}^n$

$$s \in \Sigma \to \mathbb{B}$$

We will refer to s as the *specification* and call the synthesized predicate p. The procedure optimistically assumes that the specification s can be approximated by a DNF formula over linear inequalities of the octagon domain. In the octagon domain, only zero and ± 1 are allowed factors.

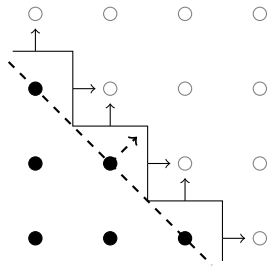
$$\pm x \cdots \pm z \leq c \pmod m$$

Unlike other template-based synthesis procedures, the number of clauses, literals per clause, and integer variables per literal is not limited a-priori.

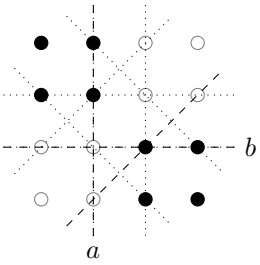
The synthesis procedure operates in two phases. During *Literal Discovery* the procedure analyzes the specification s and proposes a set of literals L. In the final *Boolean Structure Synthesis* phase, the procedure iteratively constructs conjunctive clauses that imply parts of s. The clauses are build from the literals found during Literal Discovery.

1) Literal Discovery.: The Literal Discovery phase is concerned with proposing a set of literals L that the synthesized predicate will be composed of. It exploits the relation between aliasing artifacts of the specification and the constraints underlying the specification. The generated literals are linear integer constraints.

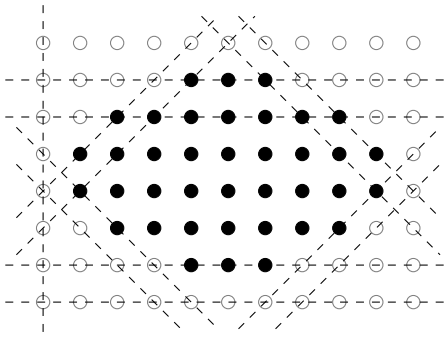
Given that the assumption holds, the literals of the formula correspond to linear inequalities of the form: $n \cdot \sigma \leq b$ where n is the normal vector and $\sigma \in \Sigma$. We will initially consider the case of a single linear inequality. Note, that the inequality does not necessarily occur in this normalized syntactic representation in the specification. Literal Discovery does not introspect into the syntactic structure of the formula. Instead, it infers the normalized representation solely by drawing point samples under constraints. The linear inequality is sampled on the integer lattice giving raise to aliasing in the image interpretation of the samples. The aliasing appears as a jagged line separating sets of points that are contained in the inequality and all the others. We overlay the jagged line with a real-valued interpretation of the inequality in Figure 12a: the jagged line loosely follows the straight line. Without introspection, we are facing the challenge of reconstructing the actual inequality from the aliasing information that is summarized in the jagged line. We shall refer to the straight line segments of the jagged line as *edges* and call sample points that have neighboring edges *border points*. We will refer to edges as feature of border points and shall make the convention that normals of edges are pointing away from the border point under consideration. In the integer lattice, two neighboring points differ only in a single component. Consequently, the normals of edges have only one non-zero component and we shall identify the normal, up to the value of that component, by calling it an edge in v. We will attribute the edges the sign of the single non-zero component. We observe the following relation between the normals of edges and the unknown normal n of the underlying linear inequality: If the normal of an edge in v is positive (negative), then $n_v < 0$ ($n_v > 0$). Importantly, there is also a point σ with edges for all v where $n_v \neq 0$. This result yields constraints on the unknown linear inequality given a border point σ and its set of edges.



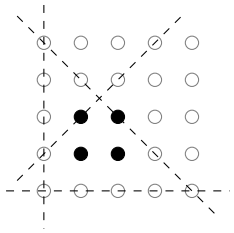
(a) Dashed: Rendering of the unknown inequality constraint with normal. Continuous: aliasing in form of a jagged line, shown with normals.



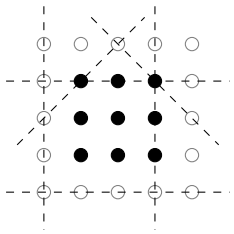
(b) Lines a and b are sufficient to partition the points. With the symmetry constraint ($y \succ x$), Literal Discovery finds the three dashed lines. Without the constraint, it will find all eight lines.



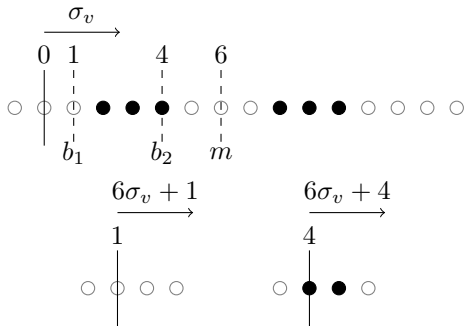
(c) Approximation of bound shapes with constraints of the octagon domain ($y \succ x$).



(d) Missing literals for small shapes due to the symmetry constraint ($y \succ x$).



(e) Artifacts superseded by correct literals in larger shapes ($y \succ x$).



(f) Above: remainder b_1 and b_2 and modulus m in variable v as returned from `DETECT_MODULUS`.Below left: sub lattice for congruence class $[b_1]$. Below right: sub lattice for congruence class $[b_2]$.

We restrict our procedure to normals with components of zero or absolute one value. This corresponds to linear inequalities of the octagon domain [3]. Coincidently, the factor restriction cancels all but one unique solution to the constraints on the unknown linear inequality. The restriction has two immediate consequences, which allow us to identify the linear inequality with a single border sample σ and its edge set: Firstly, any border point will have an edge in v, iff $n_v \neq 0$. This is because all sample points with edges are border samples, where the inequality is tight. In fact all border samples will have exactly the same set of edges and if there is no edge in v at a sample point, then $n_v = 0$. Secondly, we can rephrase the general result above specializing it for the restricted set of factors. If at a border point σ there is an edge in v with positive (negative) sign, then $n_v = -1$ ($n_v = 1$), and if there is no edge in v, then $n_v = 0$. The constant b is then $n \cdot \sigma$.

The distinction of edges by the sign of their normals reflects in the formal definition of two kinds of edges: We formally define *up* edges for negative normals and *down* edges for positive ones. Edges are defined at a position σ and in variable v[1].

$$up_v(\sigma) := s(\sigma) \oplus s(\sigma[v \to \sigma_v + 1])$$
$$down_v(\sigma) := s(\sigma) \oplus s(\sigma[v \to \sigma_v - 1])$$

[1] $\oplus$ denotes exclusive-OR.

An edge sample is given by a point σ and the set of *up* and *down* edges at that point. With the definition above, an edge sample is given by a solution to the following formula:

Input: $E \subseteq \Sigma \to \mathbb{B}$	**Output**: σ, $up_v(\sigma)$, $down_v(\sigma)$

$$\sigma \in \Sigma \tag{25}$$
$$\bigwedge_{e \in E} e(\sigma) \neq 0 \tag{26}$$
$$\bigvee_{v \in V} up_v(\sigma) \tag{27}$$
$$\bigwedge_{v \in V} \left(down_v(\sigma) \implies \bigvee_{v' \succeq v} up_{v'}(\sigma) \right) \tag{28}$$

Constraint (26) blocks edge samples that would yield already discovered inequalities. The set E controls which edge samples can be found. It invalides any edge sample that lays on lines that were constructed from earlier edge samples. Therefore, in the initial query, E is the empty set. Constraint (27) enforces that sampleσ has an edge.

A linear inequality will only be proposed for a sample σ under an order $\preceq$, iff constraint (28) holds. The performance of boolean structure synthesis phase is sensitive to the size of the literal set. This motivates the symmetry breaking constraint (28). Its effect can be seen in Figure 12b. Of the 6 proposed literals, only a and b are necessary to represent the

specification. This is because literals may be negated and so the upper left part is enclosed by the clause $a \wedge b$ and the lower right part by $\overline{a} \wedge \overline{b}$. The symmetry constraint blocks inequalities that are equivalent to already detected inequalities.

Algorithm 1 Literal Discovery main loop.

```
procedure FIND_LITERALS(spec)
    E ← ∅
    for σ ← query(E) do                                  ▷ Edge query
        e ← 0
        for v ∈ V do
            if up_v(σ) then
                e ← e + (v − σ_v)
            else if down_v(σ) then
                e ← e − (v − σ_v)
            end if
        end for
        if |E| > threshold then
            if (v, b_1, b_2, m) ← DETECT_MODULUS(E) then
                E' ← FIND_LITERALS(s(σ_v m + b_1))
                E'' ← FIND_LITERALS(s(σ_v m + b_2))
                E ← {e(σ_v m + b_1) | e ∈ E'} ∪
                    {e(σ_v m + b_2) | e ∈ E''} ∪
                    {σ_v = b_1 (mod m)} ∪
                    {σ_v = b_2 (mod m)}
            else
                abort
            end if
        end if
        E ← E ∪ {e}
    end for
    L ← {e ≤ 0 | e ∈ E}                                  ▷ Generate literals
end procedure
```

a) Sampling loop.: Our algorithm, shown in Listing 1 proposes a new linear inequality constraint for every edge sample σ, that the SMT solver finds. The state of the algorithm consists in the set E, which is comprised of lines over Σ, the set of program states.

In each iteration, the algorithm queries for a new edge sample that is not covered by the lines in E. The sample loop terminates when the set E saturates or the sampling loop seems to diverge. The algorithm assumes saturation in E if all further edge samples would lay on lines already contained in E. At this point, the last edge query becomes unsatisfiable and the algorithm terminates. Otherwise, if the size of E exceeds a given threshold, the algorithm tries to infer a modulus constraint from the lines in E.

b) Modulus detection.: `FIND_LITERALS` applies modulus detection, if the edge sampling loop seems to diverge which manifests in an excessive numbers of literals. Our prototype modulus detection will propose one modulus constraint for each variable. Modulus detection heuristically infers from the lines of E found so far a suitable modulus m and two remainders b_1 and b_2. As shown in Figure 12f above, a repeating pattern in the specification starts at b_1 and ends at b_2 modulus m.

A single modulus constraint corresponds to a repeating pattern with infinite extend. In order to find the ends of the repeating pattern, `FIND_LITERALS` recursively samples edges on the remainder classes of b_1 and b_2. We see in Figure 12f below the two sub lattices: the sub lattice for b_1 is empty. When applied to the sub lattice for b_2 `FIND_LITERALS` detects two bounding lines for the modulus constraint and returns. Note, that the threshold criterion implies that literals with modulus will be used even for large specifications that do not require them. However, this may still amount to a more compact representation in these cases.

Eventually, each line $e \in E$ is converted to a linear inequality by setting $l(\sigma) := e(\sigma) \leq 0$.

c) Completeness and Divergence.: A literal set L is complete, iff the specification can be expressed as a DNF formula over literals of L. Consequently, Literal Discovery only terminates for specifications that can be expressed with a finite literal set L. If all the required literals are in the octagon domain this is clearly the case. Otherwise, two kinds of divergence can occur: Firstly, consider the case of a single convex shape: Any bound convex shape can be approximated with constraints of the octagon domain as shown in Figure 12c. If the shape is unbound and contains a non-octagon line with infinite extend, then Literal Discovery will diverge in the attempt to approximate it with infinitely many octagon lines. Secondly, there can infinitely many convex shapes, possibly connected by superposition in a concave shape with infinite extend. In this case, our literal discovery algorithm implements partial modulus detection.

2) Boolean Structure Synthesis: During *Boolean Structure Synthesis*, the synthesis procedure incrementally constructs a DNF formula p that implies the specification s. The algorithm repeatedly constructs conjunctive clauses C, such that:

$$\forall_{\sigma \in \Sigma} \left(C(\sigma) \implies s(\sigma) \right)$$

Each constructed clause gets adjoined to p by disjunction $p \leftarrow p \vee C(\sigma)$, meaning that initially $p = \bot$.

It is known that DNF formula can be exponentially larger than the smallest possible boolean structure to represent a boolean predicate. However, the synthesis problems arising in our optimization are small and consist only of few literals and program variables. The standard compiler optimizations, such as loop peeling or unrolling, running after our optimization readily exploit the synthesized DNF formulas.

a) Cost model.: We define the cost of p to be the quality of the approximation, given by the number of assignments σ that are valid with respect to the specification s but not accepted by p:

$$cost(p) = |\{\sigma \in \Sigma | s(\sigma)\}| - |\{\sigma \in \Sigma | p(\sigma)\}|$$

This cost model motivates the design choices behind our synthesis procedure. Firstly, if a new clause C implies the specification but not the current DNF formula p, then the

cost of p decreases by adding the clause to it. Secondly, the algorithm minimizes the cost of each individual clause by only constructing prime implicants. Prime implicants are clauses, from which no literal can be dropped without violating their property that the clause implies the specification.

b) Clause construction.: Clauses are constructed from all literals found during *Literal Discovery* and any free boolean variable used in the specification. The algorithm operates by constructing an initial clause with the query:

Input: $L \subseteq \Sigma \to \mathbb{B}$	**Output**: $enable, negate \in \mathbb{B}^{\lvert L \rvert}$, σ^*

$$C(\sigma) := \bigwedge_{l \in L} (enable_l \implies negate_l \oplus l(\sigma)) \tag{29}$$

$$\forall_{\sigma \in \Sigma} (C(\sigma) \implies s(\sigma)) \tag{30}$$

$$\sigma^* \in \Sigma \wedge s(\sigma^*) \wedge C(\sigma^*) \wedge \overline{p(\sigma^*)} \tag{31}$$

Constraint 29 defines a clause C as a conjunction of selected literals from L ($enable_l$) that are optionally negated ($negate_l$). Line (30) encodes that a valid clause must be a model of the specification. As of constraint (31), the query enforces that the clause is satisfiable by at least the program state σ^*. We will refer to this program state σ^* as the *seed point*. While an initial solution clause may only hold at the seed point, clause refinement will generate more general clauses that include bigger sets of program states.

Let C' be a clause by the clause construction query. In one iteration of refinement, the algorithm tries to find a new clause C that holds at more program states than C'

$$\forall \sigma \in \Sigma . C'(\sigma) \implies C(\sigma) \tag{32}$$

$$\bigvee_{l \in C'} \left(\overline{enable_l}\right) \tag{33}$$

Here, $l \in C'$ denotes all literals that were enabled - through assignment to $enable_l$ - in the solution to the last query. The new clause C must imply the old clause C' as of line (32). Line (33) encodes the property of prime implicants, that none of their literals can be dropped without invalidating the clause.

Finally, the refinement loops terminates with an unsatisfiable formula, meaning that a prime implicant clause was found. The algorithm adds the last refined clause to the DNF formula p and queries for a new initial clause with a new seed point. If already the initial query fails, no further clause could be constructed from the given literals and the synthesis procedure returns the last DNF formula p as synthesized predicate. Otherwise, the process stops on a timeout with the last constructed DNF formula.

IV. Related Work

Finding least and greatest fixed points is not new and is, for example, discussed in the dissertation of Srivastava [4]. [5] introduce ℤ-Polyhedra, which is the intersection of polyhedra and lattices, and proof the equivalence of z-polyhedras to predicates in Presburger Arithmetic. This means our predicate template covers all predicates expressible in Presburger Arithmetic.

c) Polyhedral optimization: In the linear case, it is possible to generate the preconditions using a polyhedral model. In the polyhedral model, the optimization constraint is encoded as a union of ℤ-polyhedra. Loop invariant preconditions can be generated by eliminating iteration variables from the polyhedra. However, even state-of-the-art vectorization techniques based on the polyhedral model [6], [7] do not explicitly consider precondition synthesis. A similar effect may be achieved in some cases if the polyhedral optimizer supports loop peeling.

d) Counter-example Guided Abstraction Refinement: Both abduction procedures use ideas of Counter-Example Guided Abstraction Refinement: Literal Discovery is closely related to predicate inference. Predicate inference is involved with finding a predicate the separates a given example from a counter example. Unlike approaches based on proof interpolation, such as [8],[9], Literal Discovery uses an efficient heuristic based one edge detection to propose new predicates.

On the other hand, solving the abstract abduction problem relates to the refinement loop in CEGAR. In CEGAR, a counter example that is valid within the current abstract model is ruled out by refining the underlying lattice with predicate inference. In abstract abduction, however, we retract to more restrictive abstractions within the same lattice to rule out the counter example.

e) Pre-condition Synthesis: The problem of weakest precondition synthesis has been studied extensively in the literature, e.g. by Dijkstra [10]. Nuno et Al[11] infer preconditions for code rewriting rules. The preconditions are constraints on the read-write dependencies of the gaps in the matching pattern of the rule. If the precondition holds, the code transformation can be applied. Our approach splits the problem in two phases: Abstract Abduction, which infers a loop invariant, and Concrete Abduction, which generates a linear predicate. Dillig et al [12] discuss the synthesis of loop invariants with the `MistralSMT` solver engine. Other techniques rely on backward flow analysis to infer loop invariants and preconditions, such as in work by Cousot et al [13]. This is not directly applicable as the Divergence Analysis is a forward flow analysis. We solve the abduction problem without an explicit inverse transformer by virtue of searching the greatest abstractions on input variables that satisfy the constraints on the program. Abstract abduction solves a similar problem as in abstract debugging by Bourdoncle [14] where an early precondition for erroneous behavior later in the program is found. However, the explicit reversal of abstract interpretation has been studied extensively in the 80s, including works by Hughes et al [15].

f) Best Abstract Transformer: Abstract abduction resembles the problem of synthesizing a most precise abstract transformer as discussed in Reps et Al [16]. Their approach constructs the most precise abstract transformer pointwise for each combination of fixed abstractions on the inputs. They generate abstract transformer for a forward analysis. On the other hand, abstract abduction applied to a single operator with a lattice constraint on its output corresponds to the abstract

backward transformation of that output.

g) Counter-example Guided Synthesis: Pre-condition synthesis is a specific case of inductive synthesis, where are en explicit function is constructed from a set of implicit constraints. Many techniques implement the Counter-Example Guided Inductive Synthesis (CEGIS) paradigm, where a candidate function is refined with input/output samples. These include function templates with holes [17], [18], [19] and SMT-guided linking of operators, e.g. Jha et al [20]. Recently, Jha et al [21] presented *History Bounded Counter example Guided Inductive Synthesis* (HCEGIS), where counter-examples are weighted by their distance to the next positive sample with respect to some metric. Our edge sampling technique confines counter-examples to lay immediately next to positive examples.

There is a rich corpus of work on finding abstract interpretations of convex sets [22], [23], [24], [3], to name a few. Note, that abstract interpretation procedures only consider quantifier-free conditionals as they occurr, for example in program assertions. Our technique, however, can generate preconditions even if the specification contains quantifiers, given the underlying SMT solver is powerful enough.

A. Vectorization in presence of divergent control flow

There are two main strategies that deal with control flow divergence in vectorized programs in the literature, for example discussed in the work of Mahlke [25], A predicating vectorizer ([26]) applies if-conversion to divergent branches. Where predicated execution is not available through hardware, the vectorizer emulates it with additional fix-up code. State-of-the-art function vectorizer use a technique called Divergence Analysis, see [1], [2], to preserve as much control flow as possible in the vectorized function. For the convolution kernel, a predicating vectorizer produces code as seen in Figure 3. The predicated program does not exploit the regular behavior of traces in the image center in any way.

B. Speculative optimizations in Whole-Function Vectorization

Speculative vectorizers (Sujon et al [27]) pick paths through the program and vectorize them as straight line code. Sentinels placed at points of divergence in the paths dispatch execution to scalar code, if the assumed coherence is violated. Due to the high cost of speculation, these approaches have only been implemented in iterative compilers that rely on runtime measurements or profiling data. In our running example, speculation enables unpredicated execution of the stencil loop until the first lane diverges. This lumps the irregular address vectors at the image boundary together with the consecutive and uniform addresses in the image center, destroying the potential to optimize for the latter in the run.

Karrenberg et al [28] optimize hot code path in vectorized programs. Their experimental approach uses an SMT solver to aggregate program states that are amendable to the hot code path. They demonstrate the applicability of their technique on two benchmarks, FastWalshTransform and BitonicSort, which we revisit in our experimental section V. We supplement the synthesis procedure missing in their work and introduce a structured approach to these optimizations.

Shin et al [29] insert skipping branches in vectorized functions, that are taken if no lane will be active in a section. As their technique relies on direct tests of mask registers, the test is loop carried, if the mask is it too. In contrast, our technique is applicable for a wide-range of optimizable patterns and generates simpler conditions, that also eliminate loop dependencies by abductive reasoning.

V. Experiments

We implemented our optimization technique in the Whole-Function Vectorizer by Karrenberg [1] and used the Z3 solver by De Moura et al [30] for the SMT queries. We evaluated the runtime of the two phases of Concrete Abduction for vectorization factors 4 and 16. The number of independent integer components in the SMT encoding of the programs is shown in the last column $||\Sigma||$.

For a vectorization factor of 4, we compare the kernel runtime against the unmodified vectorizer (column *WFV*) and the OpenCL driver 2012 of Intel (column *Intel '12*). Newer versions of the latter were not supported on the machine used for runtime measurements.

The concrete abduction problems of DwtHaar1D and Prefix-Sum are identical. The same holds for the FastWalshTransform and BitonicSort benchmarks. The latter two involve non-linear integer arithmetic that neither Z3 nor other SMT solvers, such as CVC4 [31] or MiniSmT by Zankl and Middeldorp [32] could handle. However, there exists a linear predicate that could be abducted. We report how long the synthesis procedure takes to reconstruct the target solution $t \pmod 4 = 0$. The threshold for modulus detection is 100, resulting in a total of 103 SMT queries for these benchmarks.

Of the benchmarks, only Convolution2D requires Abstract Abduction.

VI. Conclusion

Abduction is a powerful tool for inferring preconditions for highly optimizable program paths. We have shown its utility for finding preconditions for non-divergent execution in vectorized functions. Despite the theoretic complexity, the SMT problems arising in this work are small and the preconditions are found quickly in the considered benchmarks.

References

[1] R. Karrenberg and S. Hack, "Whole Function Vectorization," in *Code Generation and Optimization*, 2011.

[2] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira, "Divergence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 4, pp. 13:1–13:36, Jan. 2014.

[3] A. Miné, "The octagon abstract domain," *Higher Order Symbol. Comput.*, vol. 19, no. 1, pp. 31–100, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10990-006-8609-1

[4] S. Srivastava, S. Gulwani, and J. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 497–518, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10009-012-0223-4

		SIMD factor 4							SIMD factor 16		
		Opt	WFV		Intel '12		Edges	Clauses	Edges	Clauses	$\|\Sigma\|$
Benchmark	n	[ms]	[ms]	[SpeedUp]	[ms]	[SpeedUp]	[ms] (# queries)	[ms] (# queries)	[ms] (# queries)	[ms] (# queries)	#
DwtHaar1D	2^{16}	6.24	7.90	+26.60 %	6.27	+0.48 %					
PrefixSum	512	0.077	0.119	+54.55 %	0.08	+3.90 %	6.87 (2)	14.00 (2)	8.03 (2)	15.37 (2)	2
Convolution2D		666.06	1639.47	+146.14 %	2569.46	+285.77 %	24.74 (3)	27.57 (2)	87.57 (4)	63.14 (2)	3
FastWalshTransform	2^{24}	794.74	1583.77	+99.28 %	1783.75	+124.44 %					
BitonicSort	2^{20}	584.29	576.27	-1.37 %	11625.53	+1889.68 %	979.59 (103)	16.65 (2)	1054.71 (103)	18.28 (2)	1

TABLE I: Runtime of the benchmarks and concrete abduction as measured on a Intel®Core™2 Quad CPU Q9550 machine with 2.83GHz. The CPU has SSE vector registers of width 4.

[5] G. Gupta and S. Rajopadhye, "The z-polyhedral model," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229478

[6] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 127–138. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462187

[7] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 327–337. [Online]. Available: http://dx.doi.org/10.1109/PACT.2009.18

[8] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, N. D. Jones and X. Leroy, Eds. ACM, 2004, pp. 232–244. [Online]. Available: http://doi.acm.org/10.1145/964001.964021

[9] M. N. Seghir and D. Kroening, "Counterexample guided precondition inference," in *European Symposium on Programming (ESOP)*, ser. LNCS, M. Felleisen and P. Gardner, Eds., no. 7792. Springer, 2013, pp. 451–471.

[10] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[11] N. P. Lopes and J. Monteiro, "Weakest precondition synthesis for compiler optimizations," in *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2014.

[12] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 443–456. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509511

[13] P. Cousot, R. Cousot, and F. Logozzo, "Precondition inference from intermittent assertions and application to contracts on collections," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, R. Jhala and D. Schmidt, Eds. Springer Berlin Heidelberg, 2011, vol. 6538, pp. 150–168.

[14] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 46–55. [Online]. Available: http://doi.acm.org/10.1145/155090.155095

[15] J. Hughes and J. Launchbury, "Reversing Abstract Interpretations," in *European Symposium on Programming*, ser. LNCS, vol. 582. Rennes: Springer-Verlag, 1992, also to appear in Science of Computer Programming.

[16] T. W. Reps, S. Sagiv, and G. Yorsh, "Symbolic implementation of the best transformer." in *VMCAI*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 252–266.

[17] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," Microsoft Research, Tech. Rep. MSR-TR-2008-44, March 2008.

[18] A. Solar-Lezama, "Program Synthesis by Sketching," Ph.D. dissertation, 2008. [Online]. Available: http://people.csail.mit.edu/asolar/

[19] S. Srivastava, *Satisfiability-Based Program Reasoning and Program Synthesis.* Proquest, Umi Dissertation Publishing, 2011. [Online]. Available: http://books.google.de/books?id=l6tJLgEACAAJ

[20] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806833

[21] S. Jha and S. A. Seshia, "Are there good mistakes? a theoretical analysis of cegis," in Proceedings 3rd Workshop on *Synthesis,* Vienna, Austria, July 23-24, 2014, ser. Electronic Proceedings in Theoretical Computer Science, K. Chatterjee, R. Ehlers, and S. Jha, Eds., vol. 157. Open Publishing Association, 2014, pp. 84–99.

[22] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78. New York, NY, USA: ACM, 1978, pp. 84–96. [Online]. Available: http://doi.acm.org/10.1145/512760.512770

[23] V. Laviron and F. Logozzo, "Subpolyhedra: A (more) scalable approach to infer linear inequalities," in *Verification, Model Checking, and Abstract Interpretation: Proceedings of the 10th International Conference (VMCAI 2009)*, ser. Lecture Notes in Computer Science, N. D. Jones and M. Müller-Olm, Eds., vol. 5403. Savannah, Georgia, USA: Springer-Verlag, Berlin, 2009, pp. 229–244.

[24] J. Brauer, A. King, and S. Kowalewsk, "Abstract interpretation of microcontroller code: Intervals meet congruences," *Science of Computer Programming. Methods of Software Design: Techniques and Applications*, vol. 78, issue 7, pp. 862–883, 2013.

[25] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Tech. Rep., 1996.

[26] J. C. H. Park and M. Schlansker, "On predicated execution," 1991.

[27] M. H. Sujon, R. C. Whaley, and Q. Yi, "Vectorization past dependent branches through speculation." in *PACT*. IEEE, 2013, pp. 353–362.

[28] R. Karrenberg, M. Kosta, and T. Sturm, "Presburger arithmetic in memory access optimization for data-parallel languages." in *FroCos*, ser. Lecture Notes in Computer Science, P. Fontaine, C. Ringeissen, and R. A. Schmidt, Eds., vol. 8152. Springer, 2013, pp. 56–70.

[29] J. Shin, M. W. Hall, and J. Chame, "Evaluating compiler technology for control-flow optimizations for multimedia extension architectures," *Microprocess. Microsyst.*, vol. 33, no. 4, pp. 235–243, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2009.02.002

[30] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[31] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032319

[32] H. Zankl and A. Middeldorp, "Satisfiability of non-linear (ir)rational arithmetic," in *Proceedings of the 16th International Conference on Logic for Programming and Automated Reasoning*, ser. Lecture Notes in Artificial Intelligence, vol. 6355. Dakar: Springer-Verlag, 2010, pp. 481–500.