Lookahead Widening[⋆]

Denis Gopan[1] and Thomas Reps[1,2]

[1] University of Wisconsin.
[2] GrammaTech, Inc.
{gopan,reps}@cs.wisc.edu

Abstract. We present *lookahead widening*, a novel technique for using existing widening and narrowing operators to improve the precision of static analysis. This technique is both self-contained and fully-automatic in the sense that it does not rely on separate analyzes or human involvement. We show how to integrate lookahead widening into existing analyzers with minimal effort. Experimental results indicate that the technique is able to achieve sizable precision improvements at reasonable costs.

1 Introduction

Abstract interpretation is a general framework used for static analysis and verification of software [7, 6]. In abstract interpretation, the collecting semantics of a program is expressed as a least fix-point of a set of equations. The equations are solved over some abstract domain chosen based on desired precision and cost. Typically, the equations are solved iteratively; that is, successive approximations of the solution are computed until they converge to a least fix-point. However, for many useful abstract domains (particularly those for analyzing numeric properties, such as intervals, octagons [11], and polyhedra [8, 9]) such chains of approximations can be very long or even infinite. To make use of such domains, abstract interpretation uses an extrapolation technique, called *widening* [7].

Widening attempts to predict the fix-point based on the sequence of approximations computed on earlier iterations of the analysis. Typically, widening degrades the precision of the analysis; i.e., the obtained solution is a fix-point or a post-fix-point, but not necessarily the least fix-point. If the obtained solution is a post-fix-point, it can be refined by computing a *descending approximation sequence* that converges to a (not necessarily least) fix-point. Again, the chain of approximations can be very long or infinite. To ensure convergence, either a fixed, finite number of descending iterations is performed, or a counterpart of widening, called *narrowing* [7] is used. Such use of widening and narrowing is sufficient to get precise results for loops with regular[3] behavior. However, as we illustrate in §2, it loses precision for more complex loops.

In this paper, we present a novel approach to using existing widening and narrowing operators to improve the precision of numeric program analysis. The idea behind the approach is to separate loops into phases with simpler behavior, and apply existing analysis methods to the individual phases. This offers an

[⋆] Supported by ONR(N00014-01-1-{0708,0796}) and NSF (CCR-9986308 and CCF-0524051).

[3] In this paper, we use the term *regular* in the sense of ordinary usage, i.e., consistent in action, orderly, predictable, etc. We do not use it in the mathematical sense of formal-language theory.

opportunity to obtain better results when phases of a loop have regular behavior that is lost when they are considered simultaneously. In practice, we achieve this effect by propagating two abstract values. The first value is used to keep the analysis within the current loop phase: this value is used to decide "where to go" at program conditionals and is never widened. The second value is used to compute the solution for the current phase: both widening and narrowing are applied to it. When the second value stabilizes, it is promoted into the first value, thereby allowing the analysis to advance to the next phase.

We refer to the first value as the *main value*, because it contains the overall solution after the analysis converges, and to the second value as *the pilot value*, because it "previews" the behavior of the program along the paths to which the analysis is restricted.[4] The overall technique is called *lookahead widening*, because, from the point of view of the main value, the pilot value determines a suitable extrapolation for it by sampling the analysis future.

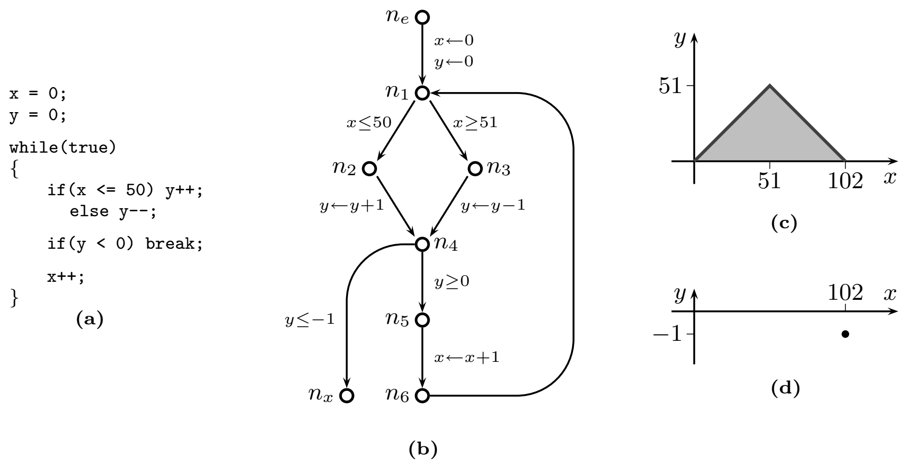
We show how to implement lookahead widening in practice so that it can be integrated into existing analyzers with minimal effort. The idea behind the implementation is to construct from an arbitrary abstract domain, such as intervals, octagons, or polyhedra, an abstract domain that implements our technique. The constructed domain can then be directly plugged into existing analyzers. However, to guarantee that such an analysis converges, we impose two minor restrictions: one on the iteration strategy employed by the analyzer, and one on the properties of the widening operator of the base domain. A major benefit of our implementation is that it can be directly used in analyzers that are not equipped for computing descending iteration sequences. Such a situation often arises when a capability to model numeric properties is added to an existing symbolic analyzer.

We present experimental results that we obtained by applying a prototype implementation of the technique to a handful of benchmarks that appeared recently in the literature on widening. Lookahead widening improves precision for half of the benchmarks. We also present experimental results from on-going work in which weighted pushdown systems are used for numeric program analysis. The use of lookahead widening in this framework allowed to establish tighter loop invariants for 4-40% of the loops in a selected set of benchmarks, with overheads ranging from 3% to 30%.

Contributions. In this paper we make the following contributions:

- We present a novel technique, lookahead widening, that uses existing widening and narrowing operators to improve the precision of static analysis. The technique is both self-contained and fully-automatic.
- We show how to implement lookahead widening in practice so that it can be integrated into existing analyzers with minimal effort.
- We present experimental results that we obtained with two prototype implementations of lookahead widening. Our results suggest that lookahead widening improves analysis precision at modest cost.

[4] The word *pilot* is used in the sense of, e.g., a sitcom pilot in the television industry.



```
x = 0;
y = 0;

while(true)
{
    if(x <= 50) y++;
      else y--;

    if(y < 0) break;

    x++;
}
```

Fig. 1. Running example: (a) a loop with non-regular behavior; (b) control-flow graph for the program in (a); (c) the set of program states at n_1: the points with integer coordinates that lie on the dark upside-down "v" are the precise set of concrete states; the gray triangle gives the best approximation of that set in the polyhedral domain; (d) the single program state that reaches n_x.

Paper organization. The paper is organized as follows: §2 introduces basic concepts and presents the running example; §3 describes lookahead widening; §4 addresses several implementation issues; §5 presents experimental results; §6 discusses related work.

2 Preliminaries

In this section, we briefly introduce several concepts that will be used throughout the paper. Due to space limitations, we assume that the reader is familiar with abstract interpretation and the standard use of widening and narrowing. For a thorough discussion of these topics, see [12, §4.2]. We assume that widening points are selected according to Bourdoncle's technique [4]. That is, we assume that a *weak topological order (WTO)* is computed for the nodes in the program's control-flow graph (CFG), and that widening is performed at the heads of the components that the WTO defines.[5] Another concept from [4] that is important for the paper is the *recursive iteration strategy*, which requires the analyzer to stabilize the currently analyzed WTO component before proceeding to CFG nodes outside of the component. The examples in the paper use the abstract domain of polyhedra [8, 9].

Running Example. We use the program in Fig. 1(a) as a running example. Fig. 2 illustrates the results from applying a solver that incorporates standard widening techniques to the program. For brevity, only some of the program points are shown. Widening is performed at node n_1 on the second and third iterations. After the third iteration, the analysis converges to a post-fix-point. A descending

[5] In structured programs, components defined by a WTO correspond to program loops.

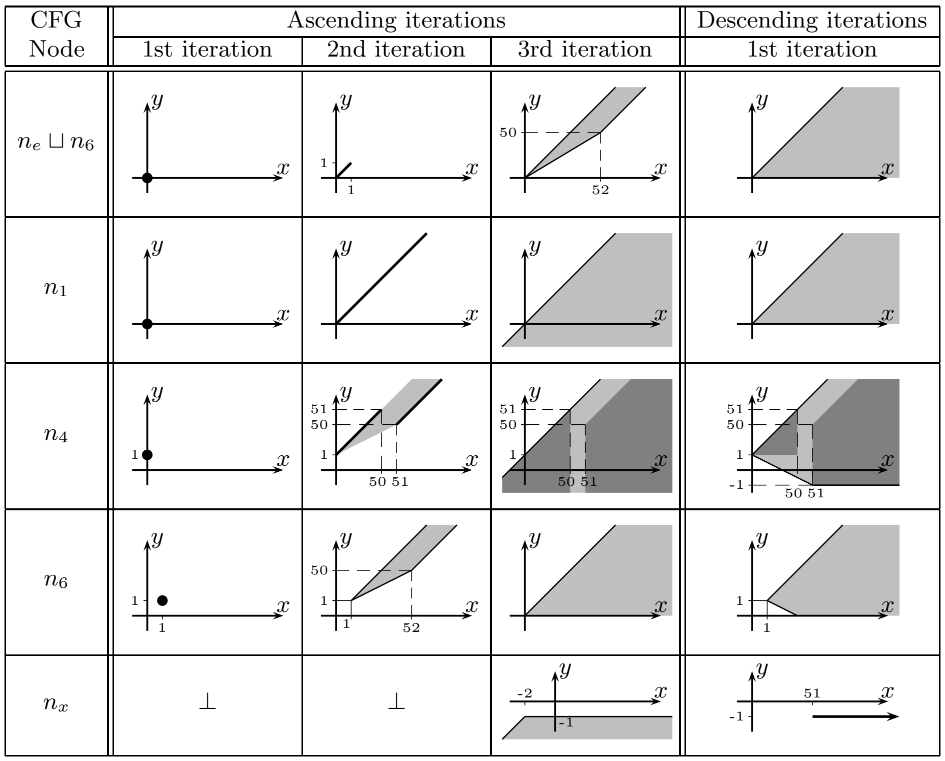


Fig. 2. Standard analysis trace. Widening is performed at node n_1. At the join point, n_4, the polyhedra that are joined are shown in dark gray and the result is shown in light gray.

iteration sequence converges in one iteration: it recovers the precision lost by the application of widening on the third iteration, but is not able to recover the precision lost by the application of widening on the second iteration.

3 Lookahead Widening

Let us start by explaining the weaknesses of the standard approach that our technique aims to overcome. The loop in Fig. 1(a) has two explicit phases: during the first phase (the first 51 iterations) both variables `x` and `y` are incremented; during the second phase (iterations 51 through 102) variable `x` is incremented, but variable `y` is decremented. While the loop's behavior during each phase is regular and could be captured precisely by standard use of widening and narrowing, the overall loop behavior is non-regular and, as was shown in §2, the standard approach yields an imprecise solution.

In particular, the limitation of the standard approach is manifested at the beginning of the second iteration. At that point in the analysis, an application of widening yields an overapproximation of the behavior of the first phase of the loop, i.e., no upper bounds are discovered for `x` and `y`. As a result, the analysis starts exploring the second phase of the loop with imprecise initial assumptions,

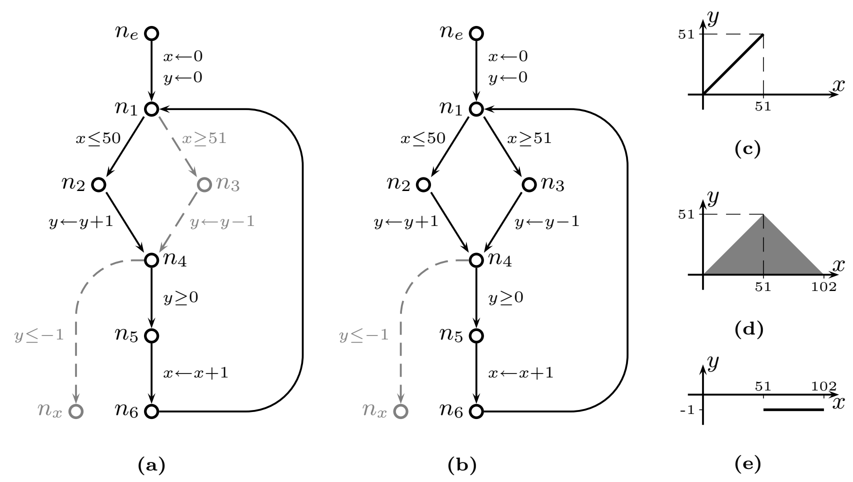


Fig. 3. Syntactic program restrictions for the program in Fig. 1: the omitted CFG nodes and edges are shown in gray and are dashed; (a) the first restriction corresponds to the first loop phase; (b) the second restriction consists of both loop phases, but not the loop exit edge; (c) solution (at n_1) for the first restriction; (d) solution (at n_1) for the second restriction; (e) solution (at n_x) for the third restriction (shown in Fig. 1(b)).

in effect, propagating the precision loss incurred on the first phase. As the last column in Fig. 2 shows, the descending iteration sequence fails to recover the lost precision.

The general idea behind our technique is to improve the precision of the analysis by obtaining a more precise solution for each loop phase before proceeding to the next. Intuitively, this can be envisioned as applying standard analysis techniques to a (finite) sequence of *syntactic restrictions* of the analyzed program that eventually converge to the entire program. The result obtained for a particular program restriction is used as the starting point for the analysis of the next restriction in the sequence.

Fig. 3 illustrates this process for our running example. Fig. 3(a) shows the first program restriction that is considered. Note that this restriction corresponds to the first phase of the loop. Fig. 3(c) shows the solution at node n_1, obtained for this restriction with the standard method. Fig. 3(b) shows the second restriction of the program; this restriction encompasses both loop phases, but does not include edges that lead outside of the loop. The analysis starts with the values obtained from the first restriction, and, at n_1, yields the solution shown in Fig. 3(d), which is the precise invariant for the loop in the polyhedral domain. The final restriction consists of the entire program (Fig. 1(b)). Applying standard analysis methods to this restriction yields, at node n_x, the solution shown in Fig. 3(e), which is the least fix-point, in the polyhedral domain, for the set of equations generated for the program in Fig. 1.

Although the above idea leads to superior results, it may seem to be hard to implement in practice. In particular, it is not obvious how to automatically derive program restrictions that correspond to loop phases. In the remainder of this section, we show how to implicitly confine the analysis to individual loop phases and how to implement our technique in a way that can be directly integrated into existing analyzers with only minor changes to their implementations.

3.1 Approximation of Loop Phases

Rather than explicitly derive syntactic program restrictions, as described above, our technique approximates this behavior by using a specially designed abstract value to guide the analysis through the program. That is, the analysis propagates a pair of abstract values: the first value (referred to as *the main value*) is used to decide at conditional points which paths are to be explored; the second value (referred to as *the pilot value*) is used to compute the solution along those paths. Widening and narrowing are only applied to the pilot value. Intuitively, the main value restricts the analysis to a particular loop phase, while the pilot value computes the solution for it. After the pilot value stabilizes, it is used to update the main value, essentially switching the analysis to the next syntactic restriction in the sequence.

Let $\mathbb{D}$ be an arbitrary abstract domain: $\mathbb{D} = \langle D, \sqsubseteq, \sqcup, \top, \bot, \nabla, \varDelta, \{\tau\}\rangle$, where D is a set of domain elements; $\sqsubseteq$ is a partial order on D; $\sqcup$, $\top$, and $\bot$ denote least upper bound operation, the greatest element, and the least element of D with respect to $\sqsubseteq$; ∇ and $\varDelta$ are the widening operator and the narrowing operator; and $\{\tau \colon D \to D\}$ is the set of (monotonic) abstract transformers associated with the edges of program's CFG. We construct a new abstract domain:

$$\mathbb{D}_{LA} = \langle D_{LA}, \sqsubseteq_{LA}, \sqcup_{LA}, \top_{LA}, \bot_{LA}, \nabla_{LA}, \{\tau_{LA}\}\rangle\,,$$

each element of which is a pair of elements of $\mathbb{D}$: one for the main value and one for the pilot value. The pilot value must either equal the main value or overapproximate it. Also, the main value (and, consequently the pilot value) cannot be bottom. We add a special element to represent bottom for the new domain:

$$D_{LA} = \{\langle d_m, d_p\rangle \mid d_m, d_p \in D,\ d_m \sqsubseteq d_p,\ d_m \neq \bot\} \cup \{\bot_{LA}\}\,.$$

The top element for the new domain is defined trivially as $\top_{LA} = \langle \top, \top\rangle$.

Abstract transformers are applied to both elements of the pair. However, to make the main value guide the analysis through the program, if an application of the transformer to the main value yields bottom, we make the entire operation yield bottom:

$$\tau_{LA}(\langle d_m, d_p\rangle) = \begin{cases} \bot_{LA} & \text{if } \tau(d_m) = \bot \\ \langle \tau(d_m), \tau(d_p)\rangle & \text{otherwise} \end{cases}$$

We define the partial order for this domain as lexicographic order on pairs:

$$\langle c_m, c_p\rangle \sqsubseteq_{LA} \langle d_m, d_p\rangle \triangleq (c_m \sqsubset d_m) \vee [(c_m = d_m) \wedge (c_p \sqsubseteq d_p)]\,.$$

This ordering allows us to accommodate a decrease in the pilot value by a strict increase in the main value, giving the overall appearance of an increasing sequence. However, the join operator induced by $\sqsubseteq_{LA}$, when applied to pairs with incomparable main values, sets the pilot value to be equal to the main value in the result. This is not suitable for our technique, because joins at loop heads, where incomparable values are typically combined, would lose all the information accumulated by pilots. Thus, we use an overapproximation of the join operator that is defined as a component-wise join:

$$\langle c_m, c_p \rangle \sqcup_{LA} \langle d_m, d_p \rangle = \langle c_m \sqcup d_m, c_p \sqcup d_p \rangle .$$

The definition of the widening operator encompasses the essence of our technique: the main value is left intact, while the pilot value first goes through an ascending phase, then through a descending phase, and is *promoted* into the main value after stabilization. Conceptually, the widening operator is defined as follows:

$$\langle c_m, c_p \rangle \nabla_{LA} \langle d_m, d_p \rangle = \begin{cases} \langle c_m \sqcup d_m, c_p \nabla d_p \rangle & \text{if the pilot value is ascending} \\ \langle c_m \sqcup d_m, c_p \mathit{\Delta} d_p \rangle & \text{if the pilot value is descending} \\ \langle d_p, d_p \rangle & \text{if the pilot value has stabilized} \end{cases}$$

The direct implementation of the above definition requires an analyzer to be modified to detect whether the pilot value is in ascending mode, descending mode, or whether it has stabilized. Also, for short phases, there is a possibility that the main value exits the phase before the pilot value stabilizes, in which case the pilot must be switched to ascending mode. These are global properties, and the modifications that are required depend heavily on the implementation of the analyzer. In our implementation, we took a somewhat different route, which we describe in the next section.

3.2 Practical Implementation

To simplify the integration of our technique into an existing analyzer, we impose on both the analyzer and the underlying abstract domain restrictions that allow us to check locally the global properties that are necessary for defining a widening operator:

- **R1. Analyzer restriction:** the analyzer must follow a *recursive iteration strategy* [4]; that is, the analysis must stay within each WTO component until the values within that component stabilize.
- **R2. Abstract domain restriction:** the abstract domain must possess a *stable widening operator* [4]; that is, $x \sqsubseteq y$ must imply that $y \nabla x = y$.

Furthermore, our implementation does not utilize narrowing operators, and only computes the equivalent of a single descending iteration for each loop phase. We believe that this simplification is reasonable because meaningful narrowing operators are only defined for a few abstract domains; also, in the experimental evaluation we did not encounter examples that would have significantly benefited from a longer descending-iteration sequences.

We define the widening operator as follows:

$$\langle c_m, c_p \rangle \nabla_{LA} \langle d_m, d_p \rangle = \begin{cases} \langle c_m, c_p \rangle & \text{if } \langle d_m, d_p \rangle \sqsubseteq_{LA} \langle c_m, c_p \rangle \\ \langle d_p, d_p \rangle & \text{if } d_p \sqsubseteq c_p \\ \langle c_m \sqcup d_m, c_p \nabla d_p \rangle & \text{otherwise} \end{cases}$$

The first case ensures that the widening operator is stable. The second case checks whether the pilot value has stabilized, and promotes it into the main value. Note that the pilot value that is promoted is not c_p, but the value d_p, which was obtained from c_p by propagating it through the loop to collect the effect of loop conditionals (i.e., one possibly-descending iteration is performed). The last case incorporates the pilot's ascending sequence: the main values are joined, and the pilot values are widened.

Soundness. It is easy to see that the results obtained with our technique are sound. Consider the operations that are applied to the main values: they precisely mimic the operations that the standard approach applies, except that widening is computed differently. Therefore, because the application of ∇_{LA} never decreases main values and because main values must stabilize for the analysis to terminate, the obtained results are guaranteed to be sound.

Convergence. We would like to show that a standard analyzer that is constructed in accordance with the principles outlined in §2 and that employs $\mathbb{D}_{LA}$ as an abstract domain converges. The use of the recursive iteration strategy (R1) allows us to limit our attention to a single WTO component: that is, if we show that the analysis converges for an arbitrary component, then it must converge for the entire program. Let us focus on the head of an arbitrary component: this is where both widening is applied and stabilization is checked.

First, we show that either the pilot value is promoted or the entire component stabilizes after a finite number of iterations. To do this, we rely on the property of the recursive-iteration strategy that the stabilization of a component can be detected by stabilization of the value at its head [4, Theorem 5]. The main value goes through a slow ascending sequence, during which time the analysis is restricted to a subset of the component's body. The pilot goes through an accelerated ascending sequence, which, if the underlying widening operator ∇ is defined correctly, must converge in a finite number of iterations. ∇_{LA} detects stabilization of the pilot's ascending sequence by encountering a first pilot value (d_p) that is less than or equal to the pilot value on the previous iteration (c_p): because the widening operator is stable (R2), application of widening will not change the previous pilot value. Note that c_p is a (post-)fix-point for the restricted component, and d_p is the result of propagating that (post-)fix-point through the same restricted component, and thus, is itself a (post-)fix-point. Two scenarios must now be considered: either the main value has also stabilized (i.e., $d_m \sqsubseteq c_m$), in which case $\langle d_m, d_p \rangle \sqsubseteq_{LA} \langle c_m, c_p \rangle$ and the entire component stabilizes (due to stability of ∇_{LA}); or the main value has not yet stabilized, in which case the (post-)fix-point d_p is promoted into the main value.

Next, we show that only a finite number of promotions can ever occur. The argument is based on the number of edges in the CFG. Depending on whether or not new CFG edges within the component's body are brought into consideration by the promotion of the pilot value into the main value, two scenarios are possible. If no new edges are brought into consideration, then the analysis stabilizes on the next iteration because both main value and pilot value are (post-)fix-points for this component. Alternatively, new CFG edges are taken

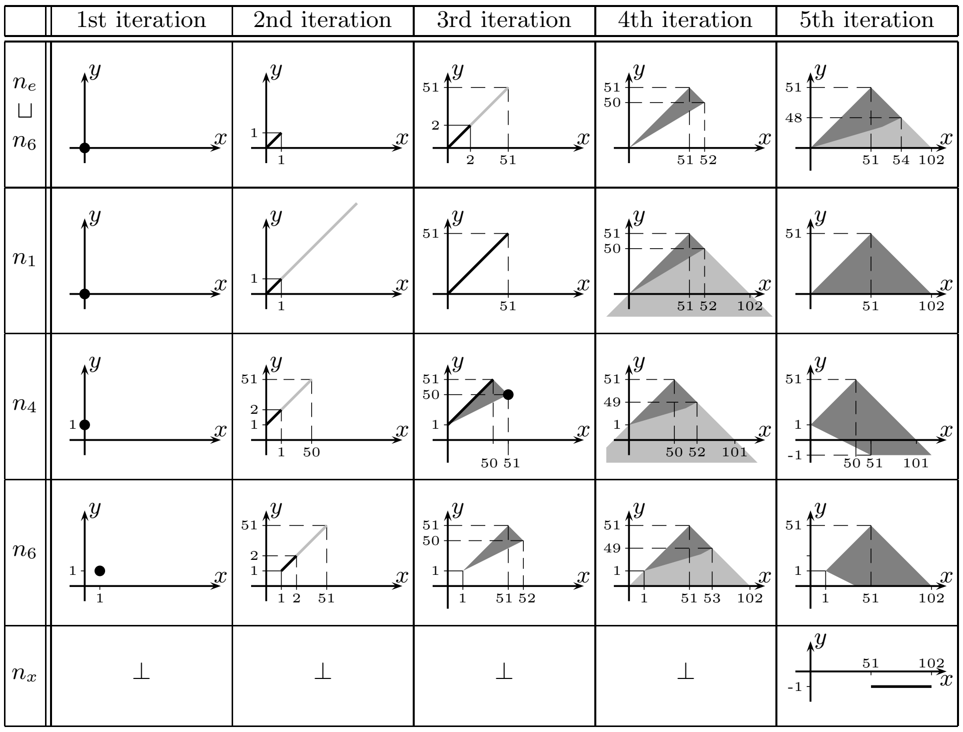


Fig. 4. Lookahead-widening analysis trace. Widening is applied at node n_1. Main values are shown in dark gray. Light gray indicates the extent of the pilot value beyond the main value. Pilot values are promoted on the 3rd and 5th iterations.

into consideration. In this case, the process described in the previous paragraph starts anew, eventually leading to the next promotion. Because the body of the component is finite, new edges can only be brought into consideration a finite number of times. Thus, there can only be a finite number of promotions before the analysis of a component converges.

Revisiting the running example. We illustrate the technique of lookahead widening by applying it to our running example. Fig. 4 shows the trace of abstract operations performed by the analysis. Due to space constraints, we only show abstract values accumulated at program points of interest. The first iteration is identical to the standard approach shown in Fig. 2. Differences are manifested on the second iteration: the widening operator propagates the unmodified main value, but applies widening to the pilot value. At node n_4, note that the pilot value has been filtered by the conditional on the edge (n_1, n_2). In contrast, in Fig. 2, the abstract state at n_4 on the second iteration has an unbounded band running off to the northeast. On the third iteration, the pilot value that reaches node n_1 is smaller than the pilot value stored there on the second iteration. Thus, this pilot value is promoted into the main value. This corresponds to the solution of the first loop phase from Fig. 3(a). As the third iteration progresses, the analysis starts exploring new CFG edges that were brought into consideration by the promotion, in essence, analyzing the program restriction from Fig. 3(b).

On the fourth iteration, at n_1, the widening operator is applied to the pilot value. At n_6, note that the pilot value has been filtered through the conditional on the edge (n_4, n_5). On the fifth iteration, the pilot value is promoted again. From here on the analysis proceeds in the same fashion as the standard analysis would, and converges on the next iteration. The analysis obtains more precise abstract values at all program points, except for n_2 (not shown in the figure).

4 Implementation Notes

"Accumulating" analyzers. In some analyzers, rather than computing the abstract value for a CFG node n as the join of the values coming from predecessors (i.e., $V(n) = \bigsqcup_{(m,n)\in E} \tau_{(m,n)}(V(m))$), such analyzers *accumulate* the abstract value at n by joining the (single) abstract value contributed by a given predecessor m_i to the value stored at n (i.e., $V(n) = V(n) \sqcup \tau_{(m_i,n)}(V(m_i))$). In particular, the WPDS++ implementation of weighted pushdown systems [10], which was our main target for integrating our technique, follows this model.

The challenge that such an analyzer design poses to lookahead widening is that the pilot value cannot be promoted directly into the main value by applying ∇_{LA} of the previous section. That is, it is not sound to update n's value by $V(n) = V(n)\nabla_{LA}\tau_{(m_i,n)}(V(m_i))$ because if the pilot value of $\tau_{(m_i,n)}(V(m_i))$ is promoted to be the main value at n, this can lose the contribution of one or more of n's other predecessors.[6] For instance, in Fig. 4, on the third iteration, an accumulating analyzer would attempt to widen the value at n_1 with the value at n_6. (The identity transformation is associated with edge (n_6, n_1).) The pilot value at n_6 is strictly smaller than the pilot value at n_1, and thus qualifies to be promoted. However, promoting it would result in an unsound main value: the point $(0, 0)$ would be excluded.

To allow lookahead widening to be used in such a setting, we slightly redefine the widening operator for accumulating analyzers. In particular, before making decisions about promotion, we join the new pilot value with the main value that is stored at the node. This makes the pilot value account for the values propagated along other incoming edges. The new widening operator is defined as follows:

$$\langle c_m, c_p\rangle \nabla_{LA} \langle d_m, d_p\rangle = \begin{cases} \langle c_m, c_p\rangle & \text{if } \langle d_m, d_p\rangle \sqsubseteq_{LA} \langle c_m, c_p\rangle \\ \langle d_p \sqcup c_m, d_p \sqcup c_m\rangle & \text{if } d_p \sqcup c_m \sqsubseteq c_p \\ \langle d_m \sqcup c_m, c_p \nabla (d_p \sqcup c_m)\rangle & \text{otherwise} \end{cases}$$

Runaway pilots. In loops (or loop phases) that consist of a small number of iterations, it is possible for the analysis to exit the loop (or phase) before the pilot value has stabilized. For instance, if the condition of the if-statement in the running example is changed to $x < 1$, the pilot value will be widened on

[6] In contrast, in analyzers that update n with the join of the values from all predecessors, any promotion of the pilot in $V(n) = V(n)\nabla_{LA} \bigsqcup_{(m,n)\in E} \tau_{(m,n)}(V(m))$ does account for the contributions from all predecessors.

the second iteration, but will not be effectively filtered through the conditionals because of the contribution from the path through node n_3, which is now enabled by the main value. As a result, the analysis will propagate a pilot value that is larger than desired, which can lead to a loss of precision at future promotions. We refer to this as the problem of *runaway pilots*.

One possible approach to alleviating this problem is to perform a promotion indirectly: that is, instead of replacing the main value with the pilot value, apply widening "up to" [9] to the main values using the symbolic concretization [13] of the pilot value as the set of "up to" constraints.

Memory usage. The abstract states shown in Fig. 4 suggest that the main value and the pilot value are often equal to each other: in our running example, this holds for abstract states that arise on the first, third, and fifth iterations of the analysis (more than half of all abstract states that arise). In our implementation, to improve memory usage, we detect this situation and store a single value instead of a pair of values when the pilot value is equal to the main value.

Delayed widening. Another interesting implementation detail is the interaction of lookahead widening with a commonly used technique called *delayed widening*. The idea behind delayed widening is to avoid applying the widening operator during the first k iterations of the loop, where k is some predefined constant. This allows the abstract states to accumulate more explicit constraints that will be used by the widening operator to generalize the loop behavior. We found it useful in practice to reset the delayed-widening counter after each promotion of the pilot value. Such resetting allows the analysis to perform k widening-free iterations at the beginning of each phase.

5 Experimental Results

We experimented with two implementations of lookahead widening: the first implementation was built into a small intraprocedural analyzer; the second implementation was built into an off-the-shelf weighted-pushdown-system solver, WPDS++ [10]. In both cases, incorporation of lookahead widening required no changes to the analysis engine.[7] Both implementations used polyhedral abstract domains built with the Parma Polyhedral Library [2].

Intraprocedural implementation. We applied the first implementation to a number of small benchmarks that appeared in recent papers about widening. The benchmarks `test*` come from work on policy iteration [5]. The `astree*` examples come from [3], where they were used to motivate *threshold widening*: a human-assisted widening technique. `Phase` is our running example, and `merge` is a program that merges two sorted arrays.

Because lookahead widening essentially makes use of one round of descending iteration for each WTO component, we controlled for this effect in our experiments by comparing lookahead widening to a slight modification of the standard widening approach: in Standard+, after each WTO component stabilizes, a single descending iteration is applied to it. This modified analysis converged for all

[7] Weighted pushdown systems, by default, do not support widening. Certain changes had to be made to the engine to make it widening-aware.

Program	Vars	Loops	Depth	Standard+		Lookahead		Overhead	Improved
				steps	LFP	steps	LFP	(% steps)	precision (%)
test1	1	1	1	19	yes	19	yes	-	-
test2	2	1	1	24	yes	24	yes	-	-
test3	3	1	1	16	-	19	-	18.8	-
test4	5	5	1	79	-	97	-	22.8	33.3
test5	2	2	2	84	yes	108	yes	28.6	-
test6	2	2	2	110	-	146	-	32.7	100.0
test7	3	3	2	93	no	104	**yes**	11.8	25.0
test8	3	3	3	45	yes	45	yes	-	-
test9	3	3	3	109	yes	142	yes	30.3	-
test10	4	4	3	227	no	266	no	17.2	20.0
astree1	1	1	1	16	no	19	**yes**	18.8	50.0
astree2	1	1	1	27	-	33	-	22.2	-
phase	2	1	1	46	no	58	**yes**	26.1	100.0
merge	3	1	1	63	no	64	**yes**	1.6	100.0

Table 1. Intraprocedural implementation results. Columns labeled *steps* indicate the number of node visits performed; *LFP* indicates whether the analysis obtains the least-fix-point solution ('-' indicates that we were not able to determine the least fix-point for the benchmark); *Improved precision* reports the percentage of *important* program points at which the analysis that used lookahead widening yielded smaller values ('-' indicates no increase in precision). Important program points include loop heads and exit nodes.

of our benchmarks, and yielded solutions that were at least as precise and often more precise than the ones obtained by the standard analysis. The only exception was `test10`, where the results at some program points were incomparable to the standard technique.

Tab. 1 shows the results we obtained. To determine least-fix-points, we ran the analysis without applying widening. The results indicate that lookahead widening achieved higher precision than the strengthened standard approach on half of the benchmarks. Also, the cost of running lookahead widening was not extremely high, peaking at about 33% extra node visits for `test6`.

Space constraints limit us to discussing just one of the benchmarks. In `astree1`, an inequation is used as the loop condition: `i = 0; while(i != 100) i++;`. We assume that the inequation '$i \neq 100$', which is hard to express in abstract domains that rely on convexity, is modeled by replacing the corresponding CFG edge with two edges: one labeled with '$i < 100$', the other labeled with '$i > 100$'. The application of widening extrapolates the upper bound for `i` to $+\infty$; the descending iterations fail to refine this bound. In contrast, lookahead widening is able to obtain the precise solution: the main value, to which widening is not applied, forces the analysis to always follow the '$i < 100$' edge, and thus the pilot value picks up this constraint before being promoted.

WPDS implementation. We used the WPDS++ implementation to determine linear relations over registers in x86 executables. CodeSurfer/x86 was used to extract a pushdown system from the executable. The contents of memory were not modeled and reads from memory were handled conservatively. Also, we

Name	Program		Push-down System				Time (sec)		Overhead	Improved
	instr	coverage	stack	same	push	pop	std	look	(%)	precision
		(%)	sym	level				ahead		(%)
speex	22364	7.9	517	483	26	20	1.13	1.33	17.4	40.0
gzip	13166	29.0	1815	2040	76	20	5.70	7.32	28.4	38.2
grep	30376	22.0	9029	10733	201	39	18.62	20.61	10.7	3.3
diff	142959	24.7	9516	11147	217	67	28.41	32.87	15.7	7.5
plot	119910	27.5	15536	15987	1050	159	44.08	45.41	3.0	20.3
graph	129040	26.0	16610	17800	824	155	53.92	56.67	5.1	19.8
calc	178378	18.7	26829	28894	1728	241	85.33	92.23	9.3	5.2

Table 2. WPDS implementation results. *Instr* lists the number of x86 instructions in the program. *Coverage* indicates what portion of each program was analyzed. *Stack symbols* correspond to program points: there are (roughly) two stack symbols per basic block. *Same-level* rules correspond to intraprocedural CFG edges between basic blocks; *push* rules correspond to procedure calls; *pop* rules correspond to procedure returns. Reported times are for the WPDS *poststar* operation. Precision improvement is given as the percentage of loop heads at which the solution was improved by the lookahead-widening technique.

chose to ignore unresolved indirect calls and jumps: as the result, only a portion of each program was analyzed. We applied this implementation to a number of GNU Linux programs that were compiled under Cygwin. The lookahead-widening technique was compared to standard widening. No descending iteration sequence was applied, because it would have required a major redesign of the WPDS++ solver. Tab. 2 presents the results obtained: lookahead widening improves the precision of the analysis on all of the benchmarks, and runs with an overhead of at most 30%.

6 Related Work

Improving widening operators [1]. One research direction is the design of more precise widening operators—that is, widening operators that are better at capturing the constraints that are present in their arguments. This approach is orthogonal to our technique: lookahead widening would benefit from the availability of more precise (base-domain) widening operators.

Widening "up to" [9] (a.k.a. *limited widening*). In this technique, each widening point is augmented with a fixed set of constraints, M. The value that is obtained from the application of the standard widening operator is further restricted by those constraints from M that are satisfied by both arguments of the widening operator. Given a well-chosen set of constraints, this technique is very powerful. A number of heuristics are available for deriving these constraint sets. In principle, the propagation of the pilot value by our technique can be viewed as an automatic way to collect and propagate such constraints to widening points. Alternatively, whenever such constraint sets are available (e.g., are derived by some external analysis or heuristic), lookahead widening can utilize them by applying widening "up to" to the pilot values. This will be beneficial when the lookahead widening is not able to break a loop into simpler phases (for instance, if a loop contains a non-deterministic conditional).

"New-control-path" heuristic [9]. This heuristic addresses imprecision that is due to new loop behaviors that appear on later loop iterations: it detects whether new paths through the loop body were explored by the analysis on its last iteration—in which case the application of widening is delayed (to let captured relationships evolve before widening is applied). While this heuristic handles the introduction of new loop behaviors well, it does not seem to be able to cope with complete changes in loop behavior, e.g., it will not improve the analysis precision for our running example. The lookahead-widening technique can be viewed as an extension of the new-control-path heuristic: not only the application of widening is delayed when the new control paths become available, but also the solution for the already explored control paths is refined by computing a descending iteration sequence.

Policy iteration [5]. This technique abandons chaotic iteration altogether in favor of different equation-solving strategies. While this technique is guaranteed to find the most precise solution, the search carried out in policy space appears to be quite expensive, and the approach requires building dedicated analyzers. In contrast, our technique can be easily integrated into existing analyzers.

References

1. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *SAS*, 2003.
2. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *SAS*, 2002.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.* Springer-Verlag, 2002.
4. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Prog. and their Appl.*, 1993.
5. A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, 2005.
6. P. Cousot. Verification by abstract interpretation. In *Symp. on Verification*, 2003.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.
9. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *FMSD*, 1997.
10. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.
11. A. Mine. The Octagon abstract domain. In *WCRE*, 2001.
12. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
13. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.