Constraint-Based Inter-Procedural Analysis of Parallel Programs

Helmut Seidl[1] and Bernhard Steffen[2]

[1] FB IV – Informatik, Universität Trier, D-54286 Trier, Germany, seidl@uni-trier.de
[2] Lehrstuhl für Programmiersysteme, Universität Dortmund, Baroper Straße 301, D-44221 Dortmund, Germany, Bernhard.Steffen@cs.uni-dortmund.de

Abstract. We provide a uniform framework for the analysis of programs with procedures and explicit, unbounded, fork/join parallelism covering not only bitvector problems like reaching definitions or live variables but also non-bitvector problems like simple constant propagation. Due to their structural similarity to the sequential case, the resulting algorithms are as efficient as their widely accepted sequential counterparts, and they can easily be integrated in existing program analysis environments like e.g. MetaFrame or PAG. We are therefore convinced that our method will soon find its way into industrial-scale computer systems.

Keywords: Inter-procedural program analysis, explicit parallelism, bitvector problems, simple constant propagation, coincidence theorems.

1 Introduction

The analysis of parallel programs is known as a notoriously hard problem. Even without procedures and with only bounded parallelism the analysis typically suffers from the so-called *state explosion problem*: in general, already the required control structures grow exponentially with the number of parallel components. Bitvector analyses, dominant in most practical compilers, escape this problem in the context of fork/join-parallelism [11, 9]: a simple pre-process is sufficient to adapt sequential intra-procedural bitvector analyses to directly work on *parallel flow graphs* which concisely and explicitly represent the program's parallelism. Key for this adaptation was to change from a *property* analysis (directly associating program points with properties) to an *effect* analysis[1] associating program points with a *property transformer* resembling the effect of the 'preceding' program fragment. The simplicity of the adaption results from the fact that bitvector analyses can conceptually be "sliced" into separate analyses for each individual bit-component each of which only requires the consideration of a three-point transformer domain.

In order to handle also procedures and unbounded parallelism, Esparza and Knoop observed that the described problem profile also admits an automata

[1] Second-order analysis in the terminology of [11].

theoretic treatment [5]. This observation has been carefully developed by Esparza and Podelski in [6]. The resulting algorithm requires involved concepts, like e.g. tree automata, and, from a program analyzer's perspective, the results are rather indirect: the reachability analysis computes regular sets characterizing the set of states satisfying a particular property. More precisely, the algorithm treats each bit-component of the analysis separately. For each such component an automata construction is required which is linear in the *product* of the size of the program and the size of an automaton describing reachable configurations. The latter automaton can grow linearly in the size of the program as well – implying that the analysis of each component is at least *quadratic* in the program size.
In this paper we present a much more direct framework for the inter-procedural analysis of fork/join parallel programs. We propose a constraint-based approach which naturally arises from an algebraic reformulation of the intra-procedural method presented in [11, 9]. Our approach closely resembles the classical understanding of bitvector analysis, has a complexity which is *linear* in the program size and admits elegant, algebraic proofs. Summarizing, we contribute to the state of the art by

1. Providing a uniform characterization of the captured analysis profile which *simultaneously* addresses all involved program entities, e.g., all program variables at once for live variable analysis or all program expressions at once for availability of expressions. Moreover, this profile goes beyond pure bitvector analyses as it e.g. also captures simple constant propagation [9].
2. Basing our development on a constraint characterization of valid parallel execution paths: the constraint system for the actual analyses simply results from an abstract interpretation [3, 4, 2] of this characterization.
3. Presenting a framework which supports algebraic reasoning. E.g., the proof for proposition 2(3) – resembling the central Main Lemma of [11] – straightforwardly evolves from our profile characterization.
4. Guaranteeing essentially the same performance as for purely inter-procedural bitvector analyses by exploiting the results of a generalized *possible interference* analysis [11].

As a consequence, the presented framework is tightly tailored for the intended application area. It directly associates the program points with the required information based on classical constraint solving through (e.g., worklist based) fixpoint iteration. This can be exploited to obtain simple implementations in current program analysis generators like DFA& OPT MetaFrame [10] or PAG [1], which provide all the required fixpoint iteration machinery.
The paper is organized as follows. After formally introducing explicitly parallel programs with procedures in section 2, we define the notion of parallel execution paths in section 3, and specify our analysis problem in section 4. Section 5 then presents a precise effect analysis for procedures, which is the basis for the precise inter-procedural reachability analysis given in section 6. Finally, section 7 discusses possible extensions of our formal development, while section 8 gives our conclusions and perspectives.

2 Programs as Control-flow Graphs

We assume that programs are given as (annotated) control-flow graphs (cfg's for short). An edge in the cfg either is a call of a single procedure, a parallel call to two procedures, or a basic computation step. An example of such a cfg is given in figure 1. There, we only visualized the annotation of call and parallel call edges. Observe that this cfg indeed introduces an unbounded number of instances of procedure q running in parallel.

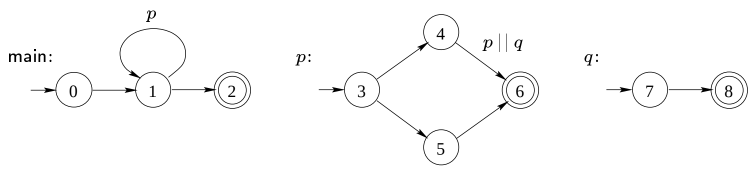

Fig. 1. An Example Control-flow Graph.

Formally, a control-flow graph $\mathcal{G}$ for a program with procedures and explicit parallelism consists of a finite set Proc of procedures together with a collection $\mathcal{G}_p, p \in \mathsf{Proc}$, of disjoint *intra-procedural* control-flow graphs. We assume that there is one special procedure main with which program execution starts. The intra-procedural control-flow graph $\mathcal{G}_p$ of a procedure p consists of:

- A set N_p of *program points*;
- A special *entry point* $s \in N_p$ as well as a special *return point* $r \in N_p$;
- A set of edges $E_p \subseteq N_p \times N_p$;
- A subset $C_p \subseteq E_p$ of *call edges* where for $e \in C_p$, $\mathsf{call}\, e = p$ denotes that edge e calls the procedure p; and finally,
- A subset $P_p \subseteq E_p$ of *parallel call edges* where for $e \in P_p$, $\mathsf{call}\, e = p_1 \,||\, p_2$ denotes that edge e calls the procedures p_1 and p_2 in parallel.

Edges which are not contained in C_p or P_p are also called *basic edges*.

Practical Remark: It is just for convenience that we allow only binary parallelism in our programs. Our methods can be easily adapted to work also for more procedures being called in parallel or even parallel do-loops.
Also note that we do not consider synchronization between parallel threads by barriers or semaphores. Such constructs *limit* the amount of possible execution paths. By ignoring these, we may get *more* possible execution paths and thus (perhaps less precise but) still *safe* analysis results.

3 Parallel Execution Paths

The semantics of a parallel program is determined w.r.t. the set of parallel execution paths. What we are now going to formalize is an *interleaving semantics*

for parallely executable threads. We need the following auxiliary definitions. Let E denote a finite set of edges. Let $w = e_1 \dots e_n$ be a word from E^* and $I = \{i_1 < \dots < i_k\} \subseteq \{1, \dots, n\}$ be a subset of positions in w. Then the *restriction* of w to I is given by $w|_I = e_{i_1} \dots e_{i_k}$.
The *interleaving* of subsets $M_1, M_2 \subseteq E^*$ is defined by

$$M_1 \otimes M_2 = \{w \in E^* \mid \exists I_1 + I_2 = \{1, \dots, |w|\} : w|_{I_1} \in M_1 \text{ and } w|_{I_2} \in M_2\}$$

Here, "+" denotes disjoint union of sets. Thus, $M_1 \otimes M_2$ consists of all possible interleavings of sequences from M_1 and M_2. Furthermore for $M \subseteq E^*$, let $\mathrm{pre}(M)$ denote the set of all prefixes of words in M, i.e.,

$$\mathrm{pre}(M) = \{u \in E^* \mid \exists v \in E^* : uv \in M\}$$

We consider the following sets of possible execution paths:

- For $p \in \mathsf{Proc}$, the set $\Pi(p)$ of all execution paths for p;
- For program point v of procedure p, the set $\Pi(v)$ of all paths starting at the entry point of p and reaching v on the *same level* (see below);
- For every procedure p, the set $\Pi_r(p)$ of all paths starting at from a call of main and *reaching* some call of p;
- For every program point v, the set $\Pi_r(v)$ of all paths starting at from a call of main and *reaching* program point v.

These sets are given through the least solutions of the following constraint systems (whose variables for simplicity are denoted by $\Pi(p), \Pi(v), \Pi_r(p), \Pi_r(v)$ as well). Let us start with the defining constraint system for the sets of same-level execution paths.

$\Pi(p) \supseteq \Pi(r)$	r return point of p	(1)
$\Pi(s) \supseteq \{\epsilon\}$	s entry point of a procedure	(2)
$\Pi(v) \supseteq \Pi(u) \cdot \{e\}$	$e = (u, v)$ basic edge	(3)
$\Pi(v) \supseteq \Pi(u) \cdot \Pi(p)$	$e = (u, v)$ calls p	(4)
$\Pi(v) \supseteq \Pi(u) \cdot (\Pi(p_1) \otimes \Pi(p_2))$	$e = (u, v)$ calls $p_1 \,\|\, p_2$	(5)

Lines (1) through (4) are the standard lines to determine the sets of all same-level execution paths as known from inter-procedural analysis of sequential programs. Line (1) says that the set of execution paths of procedure p is the set of same-level paths reaching the return point of p. Line (2) says that at least ϵ is a same-level execution path that reaches the entry point of a procedure. Line (3) says that for every basic edge $e = (u, v)$, the set of same-level execution paths reaching the program point v subsumes all same-level execution paths to u extended by e. Line (4) says that for every edge $e = (u, v)$ calling a procedure p, the set of same-level execution paths reaching the program point v subsumes all same-level execution paths reaching u extended by any execution path through the procedure p. Line (5) for a parallel call of $p_1 \,||\, p_2$ has the same form as line (4). But now the same-level execution paths to the program point before the call are extended by all interleavings of execution paths for p_1 and p_2.

In order to specify the sets $\Pi_r(p), \Pi_r(v)$, let us introduce the auxiliary sets $\Pi(v,p)$, v a program point, p a procedure, which give the sets of execution paths reaching v from a call of p. These auxiliary sets are defined as the least solution of the following system of constraints:

$$
\begin{array}{lll}
\Pi(v,q) \supseteq \Pi(v) & v \text{ program point of procedure } q & (1)\\
\Pi(v,q) \supseteq \Pi(u) \cdot \Pi(v,p) & e = (u, \_) \text{ calls } p \text{ in } q & (2)\\
\Pi(v,q) \supseteq \Pi(u) \cdot (\Pi(v,p_i) \otimes M) & e = (u, \_) \text{ calls } p_1 \,||\, p_2 \text{ in } q & (3)
\end{array}
$$

where M in line (3) is given by $M = \mathrm{pre}(\Pi(p_{3-i}))$. The intuition behind this definition is as follows. Line (1) says that whenever v is a program point of procedure q, then the set of execution paths from q to v subsumes all same-level execution paths from q to v. Line (2) says that whenever at some edge $e = (u, \_)$ in the body of procedure q, some procedure p is called, then the set of execution paths from q to v subsumes all computation paths consisting of a same-level execution path from q to the program point u followed by an execution path from p to v. Finally, line (3) considers an edge $e = (u, \_)$ in the body of q which is a parallel call of p_1 and p_2. Then we have to append to the same-level execution paths to u all interleavings of execution paths from p_i to v with prefixes of same-level execution paths for the parallel procedure.
Given the $\Pi(v,q)$, we define the values $\Pi_r(v), \Pi_r(p)$ as the least solution of:

$$
\begin{array}{ll}
\Pi_r(v) \supseteq \Pi(v, \mathsf{main}) & v \text{ a program point}\\
\Pi_r(p) \supseteq \Pi_r(u) & \text{edge } (u, \_) \text{ calls } p,\ p \,||\, \_ \text{ or } \_ \,||\, p
\end{array}
$$

For now, let us assume that all the sets of execution paths $\Pi(v), \Pi_r(v), \Pi(p), \Pi_r(p)$ are non-empty. In section 7 we will explain how this assumption can be removed.

4 Semantics

Let $\mathbb{D}$ denote a complete lattice and $\mathbb{F} \subseteq \mathbb{D} \to \mathbb{D}$ a subset of monotonic functions from $\mathbb{D}$ to $\mathbb{D}$ which contains $\lambda x.\bot$ (the constant $\bot$-function) and $I = \lambda x.x$ (the identity) and is closed under composition "$\circ$" and least upper bounds. While $\mathbb{D}$ is meant to specify the set of abstract properties, $\mathbb{F}$ describes all possible ways how properties may be transformed when passing from one program point to the other. In this paper we make the following additional assumption:

- $\mathbb{D}$ is *distributive*, i.e., $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$ holds for all $a, b, c \in \mathbb{D}$;
- $\mathbb{D}$ has height $h < \infty$, i.e., every ascending chain of elements in $\mathbb{D}$ has length at most $h + 1$;
- set $\mathbb{F}$ consists of all functions of the form $f\,x = (a \sqcap x) \sqcup b$ with $a, b \in \mathbb{D}$.

Since $\mathbb{D}$ is distributive, all functions f in $\mathbb{F}$ are distributive as well, i.e., $f\,(a \sqcup b) = (f\,a) \sqcup (f\,b)$ for all $a, b \in \mathbb{D}$. Let us also mention that neither $\mathbb{D}$ nor $\mathbb{F}$ is demanded to be finite. However, since $\mathbb{D}$ has height h, the lattice $\mathbb{F}$ has height at most $2h$. The most prominent class of problems that satisfy our restrictions are *bitvector problems* like available expressions, reaching definitions, life variables or very busy expressions [7]. In these cases, we may choose $\mathbb{D} = \mathbb{B}^h$ where $\mathbb{B} = \{0 \sqsubset 1\}$.

There are, however, further analysis problems which meet our assumptions without being bitvector problems. This is the case, e.g., for simple constant propagation. Simple constant propagation tries to determine whether or not some constant has been assigned to a variable which later-on remains unchanged. For this application, we may choose $\mathbb{D} = V \to \mathbb{B}$ where V is the set of program variables and $\mathbb{B}$ is the flat lattice of possible values for program variables. Thus, an abstract value $d \in \mathbb{D}$ represents an assignment of variables to values. In particular, $\mathbb{D}$ has height $h = 2 \cdot \#V$. Note furthermore that for simple constant propagation, all functions $f \in \mathbb{F}$ are of the special form $f = \lambda x.(a \sqcap x) \sqcup b$ with $a \in \{\bot, \top\}$. Thus, ascending chains of functions have length at most $3 \cdot \#V$.
Let E denote a set of edges and $[.] : E \to \mathbb{F}$ denote an assignment of functions to edges. Then we extend $[.]$ to sequences $w = e_1 \dots e_n \in E^*$ and sets $M \subseteq E^*$ in the natural way, i.e., by

$$[w] = [e_n] \circ \dots \circ [e_1] \qquad\qquad [M] = \bigsqcup\{[w] \mid w \in M\}$$

Thus, especially, $[\emptyset] = \lambda x.\bot$ (the least element in $\mathbb{F}$), and $[\{\epsilon\}] = [\epsilon] = I$. Functions $[w]$ and $[M]$ are also called the *effect* of the sequence w and the set M, respectively.
For the rest of this paper we assume that we are given an assignment

$$[e] = f_e = \lambda x.(a_e \sqcap x) \sqcup b_e \in \mathbb{F}$$

to each basic edge e of our input program. Then program analysis tries to compute (approximations to) the following values:

Effects of Procedures: For each procedure p, $\mathsf{Effect}(p) := [\Pi(p)]$ denotes the effect of the set of all same-level execution paths through p;

Reachability: For a start value $d_0 \in \mathbb{D}$, program point v and procedure p, $\mathsf{Reach}(v) := [\Pi_r(v)]\, d_0$ and $\mathsf{Reach}(p) := [\Pi_r(p)]\, d_0$ denote the least upper bounds on all abstract values reaching v along execution paths from main and the least upper bound on all abstract values reaching calls to p, respectively.

The system of these values is called the *Merge-Over-all-Paths* solution (abbreviated: MOP solution) of the analysis problem. Since the respective sets of execution paths are typically infinite, it is not clear whether this solution can be computed effectively. The standard approach proposed in data-flow analysis and abstract interpretation [3, 4, 2] consists in putting up a set $\mathcal{C}$ of constraints on the values we are interested in. The constraints are chosen in such a way that any solution to $\mathcal{C}$ is guaranteed to represent a safe approximation of the values. Quite frequently, however, the least solution of $\mathcal{C}$ equals the MOP solution [8, 13]. Then we speak of *coincidence* of the solutions, meaning that $\mathcal{C}$ *precisely* characterizes the MOP.
In our present application, we are already given a constraint system whose least solution represents the sets of execution paths which are to be evaluated. By inspecting this constraint system, we would naturally try to obtain constraint systems for effect analysis and reachability just by abstracting the lattice of sets

of paths with our lattice $\mathbb{F}$. Thus, the ordering "$\subseteq$" of set inclusion on sets of paths is mapped to the ordering on $\mathbb{F}$; set union and concatenation is mapped to least upper bounds and composition of functions. Indeed, this abstraction mapping [.] has the following properties:

Proposition 1. *Let $M_1, M_2 \subseteq E^*$. Then the following holds:*

1. $[M_1 \cup M_2] = [M_1] \sqcup [M_2]$;
2. $[M_1 \cdot M_2] = [M_2] \circ [M_1]$ *if both M_1 and M_2 are non-empty.* □

Proposition 1 suggests a direct translation of the constraint system for the sets of execution paths into a constraint system which we are aiming at. The only two obstacles withstanding a direct translation are (1) an abstract interleaving operator (which for simplicity is denoted by "$\otimes$" as well), and (2) a way how to deal with prefixes. For our abstract lattices, these two problems turn out to have surprisingly simple solutions.
For $f_i = \lambda x.(a_i \sqcap x) \sqcup b_i$, $i = 1, 2$, we define the *interleaving* of f_1 and f_2 by:

$$f_1 \otimes f_2 = \lambda x.(a_1 \sqcap a_2 \sqcap x) \sqcup b_1 \sqcup b_2$$

We have:

Proposition 2. *Let $f_1, f_2, f \in \mathbb{F}$. Then the following holds:*

1. $f_1 \otimes f_2 = f_1 \circ f_2 \sqcup f_2 \circ f_1$;
2. $(f_1 \sqcup f_2) \otimes f = f_1 \otimes f \sqcup f_2 \otimes f$;
3. $[M_1 \otimes M_2] = [M_1] \otimes [M_2]$ *for non-empty subsets $M_1, M_2 \subseteq E^*$.*

For a proof of Proposition 2 see appendix A. Let us now consider the set pre(M) of prefixes of a non-empty set $M \subseteq E^*$. Then the following holds:

Proposition 3. *Let E_M denote the edges occurring in elements of M where for $e \in E_M$, $[e] = \lambda x.(a_e \sqcap x) \sqcup b_e$. Then*

$$[pre(M)] = \lambda x.x \sqcup B \qquad \text{where} \qquad B = \bigsqcup\{b_e \mid e \in E_M\}$$ □

Thus, all the intersections with the a_e have disappeared. What only remains is the least upper bound on the values b_e.

5 Effect Analysis

Now we have all prerequisites together to present a constraint system for effect analysis. The least solution of the constraint system defines values $[p]$ for the effect of procedures p together with values $[v]$ for the effects of same-level execution paths reaching program point v.

$[p] \sqsupseteq [r]$	r return point of p	(1)
$[s] \sqsupseteq I$	s entry point	(2)
$[v] \sqsupseteq f_e \circ [u]$	$e = (u, v)$ basic edge	(3)
$[v] \sqsupseteq [p] \circ [u]$	$e = (u, v)$ calls p	(4)
$[v] \sqsupseteq ([p_1] \otimes [p_2]) \circ [u]$	$e = (u, v)$ calls $p_1 \mid\mid p_2$	(5)

Lines (1) through (4) are the lines to determine the effects of procedures as known from inter-procedural analysis of sequential programs. Line (1) says that the effect of procedure p is the effect of what has been accumulated for the return point of p. Line (2) says that accumulation of effects starts at entry points of procedures with the identity function $I = \lambda x.x$. Line (3) says that the contribution of a basic edge $e = (u, v)$ to the value for v is given by the value for u extended by the application of the function f_e associated with this edge. Line (4) says that the contribution of an edge $e = (u, v)$ calling a procedure p is determined analogously with the only difference that the function f_e in line (3) is now replaced with the effect $[p]$ of the called procedure. Also line (5) for a parallel call has the same form. But now, in order to determine the combined effect of the parallely executed procedures p_1 and p_2, we rely on the interleaving operator "$\otimes$". This constraint system for effect analysis is the direct abstraction of the corresponding constraint system for same-level reaching paths from section 3. Therefore, we obtain (by distributivity of all involved operators):

Theorem 1. *The least solution of the effect constraint system precisely describes the effect of procedures, i.e.,*

$$\mathsf{Effect}(p) = [p] \qquad \textit{and} \qquad \mathsf{Effect}(v) = [v]$$

for every procedure p and program point v. These values can be computed in time $\mathcal{O}(h \cdot n)$ where n is the size of the program. □

6 A Constraint System for Reachability

As for effect analysis, we could mimic the least fixpoint definition of the sets of reaching execution paths through a corresponding constraint system over $\mathbb{F}$. Observe, however, that our defining constraint system for reaching execution paths in section 3 has quadratic size. Clearly, we would like to improve on this, and indeed this is possible – even without sacrificing precision.

Instead of accumulating effects in a topdown fashion as was necessary in the precise definition of reaching execution paths, we prefer a bottom-up accumulation – a strategy which is commonly used in inter-procedural analysis of sequential programs. There, accumulation directly starts at the main program and then successively proceeds to called procedures.

For each program point v, let $B(v)$ denote the least upper bound of all b_e, for all basic edges e possibly executed in parallel with v. This value is also called *possible interference* of v. Formally, these values are determined through the least solution of the following constraint system:

$$\begin{array}{lll}
\sigma(p) & \sqsupseteq b_e & e \text{ basic edge in procedure } p \\
\sigma(p) & \sqsupseteq \sigma(q) & \text{procedure } p \text{ calls } q \text{ or } q \,||\, _ \text{ or } _ \,||\, q \\
B(v) & \sqsupseteq B(p) & v \text{ program point in } p \\
B(p) & \sqsupseteq B(u) & (u, _) \text{ calls procedure } p \\
B(q_i) & \sqsupseteq \sigma(q_{3-i}) \sqcup B(u) & (u, _) \text{ calls } q_1 \,||\, q_2
\end{array}$$

We used auxiliary values $\sigma(p)$, p a procedure, to calculate the least upper bound on b_e for all basic edges possibly executed during evaluation of p. The whole system for computing the values $\sigma(p)$, $B(p)$ and $B(v)$ is of linear size and uses "$\sqcup$" as only operation in right-hand sides. Such kind of problems are also known as "pure merge problems" and can be solved even in linear time.
We will now construct a constraint system as for inter-procedural reachability analysis of sequential programs, but for each program point additionally take its possible interference into account. Thus, we consider the values $[\![v]\!]$, v a program point, $[\![p]\!]$, p a procedure, which are determined as the least solution of the following constraint system:

$$\begin{array}{llll}
[\![\mathsf{main}]\!] & \sqsupseteq d_0 & & (1) \\
[\![v]\!] & \sqsupseteq B(v) & \text{and} & (2) \\
[\![v]\!] & \sqsupseteq [v][\![p]\!] & v \text{ program point in procedure } p & (3) \\
[\![p]\!] & \sqsupseteq [\![u]\!] & e = (u, _) \text{ calls } p \text{ or } p \,||\, _ \text{ or } _ \,||\, p & (4)
\end{array}$$

Only line (2) makes the difference to a corresponding constraint system for reachability in sequential programs. The intuition behind the constraint system is as follows. Line (1) says that initially the value reaching main should subsume the initial value d_0. Line (2) says that the value reaching program point v should subsume its possible interference. Line (3) says that when v is a program point of procedure p, then the reaching value should also subsume the intra-procedural effect of v applied to the value reaching p. Line (4) finally says that the value reaching a procedure should subsume the value of every program point where such a call (possibly in parallel to another call) is possible.
This constraint system differs considerably from the constraint system for the sets of reaching execution paths. Nonetheless, we are able to prove:

Theorem 2. *The above constraint system computes precise reachability information as well, i.e.,*

$$\mathsf{Reach}(p) = [\![p]\!] \qquad \textit{and} \qquad \mathsf{Reach}(v) = [\![v]\!]$$

for all program points v and procedures p. These values can be computed in time $\mathcal{O}(h \cdot n)$ where n is the size of the program.

For a proof see appendix B. Theorem 2 implies that programs with procedures and parallelism are not harder to analyze than programs with procedures but without parallelism!

7 Extensions

In this section, we discuss issues which are important for the practical applicability of the presented results. We do not claim that this section contains any new ideas or constructions. Rather we want to emphasize that the constructions known from the inter-procedural analysis of sequential programs can be extended to parallel programs in a straight-forward way.

7.1 Non-Reachable Program Points

So far, we assumed that every program point is reachable by at least one execution path. In order to show that this assumption is not vital, let $\mathcal{P}$ and $\mathcal{R}$ denote the sets of possibly terminating procedures and reachable program points, respectively. In order to compute these sets, we instantiate our generic analysis with $\mathbb{D} = \{0 \sqsubset 1\}$ where for each basic edge e, the function $[e] = f_e$ is given by $f_e = I = \lambda x.x$, and the initial value d_0 equals 1. The only functions from $\mathbb{D} \to \mathbb{D}$ occurring during the analysis are $\lambda x.\bot$ and I. Both functions are *strict*, i.e., map $\bot$ to $\bot$. Therefore, we obtain:

Proposition 4. *For every procedure p and program point v, the following holds:*

1. $[v] = I$ *iff* $\Pi(v) \neq \emptyset$ *and* $[p] = I$ *iff* $\Pi(p) \neq \emptyset$*;*
2. $[\![v]\!] = 1$ *iff* $\Pi_r(v) \neq \emptyset$ *and* $[\![p]\!] = 1$ *iff* $\Pi_r(p) \neq \emptyset$*.*

In particular, $p \in \mathcal{P}$ iff $[p] = I$, and $v \in \mathcal{R}$ iff $[\![v]\!] = 1$. □

We conclude that the sets $\mathcal{P}$ and $\mathcal{R}$ can be computed in linear time.

A non-reachable program point should not influence any other program point. Therefore, we modify the given cfg by removing all edges starting in program points not in $\mathcal{R}$. By this edge removal, the sets of reaching execution paths have not changed. Let us call the resulting cfg *normalized*. Then we obtain:

Theorem 3. *Assume the cfg is normalized. Then for every program point v and procedure p,*

1. $\mathsf{Effect}(v) = [v]$ *and* $\mathsf{Effect}(p) = [p]$*;*
2. $\mathsf{Reach}(v) = [\![v]\!]$ *and* $\mathsf{Reach}(p) = [\![p]\!]$*.* □

We conclude that, after the preprocessing step of normalization, our constraint systems will compute a safe approximation which is precise.

Practical Remark: Normalization of the cfg may remove edges and thus some constraints from the constraint systems of the analysis. Therefore, omitting normalization may result in a less precise, but still safe analysis.

7.2 Backward Analysis

What we discussed so far, is called *forward analysis*. Examples of forward analysis problems are reaching definitions, available expressions or simple constant propagation. Other important analyses, however, determine the value at a program point v w.r.t. the *possible future* of v, i.e., the set of reverses of execution paths possibly following a visit of v. Examples are live variables or very busy expressions. Such analyses are called *backward analyses*. In case that every forward reachable program point is also backward reachable, i.e., lies on an execution path from the start point to the return point of `main`, we can reduce backward analysis to forward analysis – simply by normalizing the cfg followed by a reversal of edge orientations and an exchange of entry and return points of procedures.

7.3 Local and Global State

Consider an edge $e = (u, v)$ in the cfg which calls a terminating procedure p (the treatment of a terminating parallel call to two procedures p_1 and p_2 is completely analogous). So far, the complete information at program point u is passed to the entry point of p. Indeed, this is adequate when analyzing *global* properties like availability of expressions which depend on global variables only. It is not (immediately) applicable in presence of *local* variables which are visible to the caller but should be hidden from the callee p, meaning that they should survive the call unchanged [8, 13].

To make things precise, let us assume that $\mathbb{D} = \mathbb{D}_l \times \mathbb{D}_g$ where $\mathbb{D}_l$ and $\mathbb{D}_g$ describe local and global properties, respectively. Let us further assume that the global part of the current state is passed as a parameter to p, and also returned as the result of the call, whereas the local part of the program point before the call is by-passed the call using some transformer $\beta_e : \mathbb{D}_l \to \mathbb{D}_l$. Recall that every $f \in \mathbb{F}$ is of the form $f = \lambda x.(x \sqcap a) \sqcup b$ with $a, b \in \mathbb{D}$. Since $\mathbb{D}$ is a Cartesian product, this implies that $f = f_l \times f_g$ where $f_l : \mathbb{D}_l \to \mathbb{D}_l$ and $f_g : \mathbb{D}_g \to \mathbb{D}_g$ independently operate on the local states and global states, respectively.

Therefore, we can separate the analysis into two phases.

The first phase considers just global values from $\mathbb{D}_g$. No local state need to be preserved during the call, and we use the original call edge.

The second phase then is purely intra-procedural and deals with the lattice $\mathbb{D}_l$. But now, since the call at edge e has no effect onto the local state, we simply change e into a basic edge with $[e] = \beta_e$.

8 Conclusion and Perspectives

We have shown how to extend the intra-procedural method of [11] to uniformly and efficiently capture inter-procedural bitvector analyses of fork/join parallel programs. Our method, which comprises analysis problems like available expressions, live variables or simple constant propagation, passes the test for practicality, as it 'behaves' as the widely accepted algorithms for sequential inter-procedural program analysis. Moreover, even though precision can only be proved for fork/join parallelism, our algorithm may also be used for computing safe approximations for languages with arbitrary synchronization statements. Finally, due to its structural similarity to the sequential case, it can easily be integrated in program analysis environments like e.g. MetaFrame or PAG, which already contain the necessary fixpoint machinery.

As a next step, we plan a closer comparison with the automata theoretic approach of Esparza and Podelski. The considered program structures are obviously similar, however, the range of possible analyses may be different. As shown in [12], the automata theoretic approach is able to capture the model checking problem for all of the linear time temporal logic *EF*. It would be interesting to see whether it is possible to adopt our technique to covering this logic as well, or whether the automata theoretic approach, which is significantly more complex already for the analysis problem considered here, is inherently more powerful.

References

1. Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of 2nd Static Analysis Symposium (SAS)*, pages 33–50. LNCS 983, Springer Verlag, 1995.
2. Patrick Cousot. Semantic Foundations of Program Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
3. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.
4. Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.
5. J. Esparza and J. Knoop. An Automata-theoretic Approach to Interprocedural Data-flow Analysis. In *FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science (LNCS)*, pages 14–30. Springer-Verlag, 1999.
6. J. Esparza and A. Podelski. Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs. In *ACM International Conference on Princples of Programming Languages (POPL)*, 2000. To appear.
7. M.S. Hecht. *Flow Analysis of Computer Programs.* The Computer Science Library. North-Holland, New York, 1977.
8. J.Knoop and B.Steffen. The Interprocedural Coincidence Theorem. In *4th International Conference on Compiler Construction (CC'92)*, volume 641 of *Lecture Notes in Computer Science (LNCS)*, pages 125–140. Springer-Verlag, 1992.
9. J. Knoop. Parallel Constant Propagation. In *4th European Conference on Parallel Processing (Euro-Par)*, volume 1470 of *Lecture Notes in Computer Science (LNCS)*, pages 445–455. Springer-Verlag, 1998.
10. J. Knoop, O. Rüthing, and B. Steffen. Towards a Tool Kit for the Automatic Generation of Interprocedural Data Flow Analyses. *Journal of Programming Languages*, 4(4):211–246, December 1996. Chapman & Hall, London (UK).
11. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.
12. D. Lugiez and P. Schnoebelen. The Regular Viewpoint on PA-Processes. In *9th International Conference on Concurrency (CONCUR)*, volume 1466 of *Lecture Note In Computer Science (LNCS)*, pages 50–66. Springer-Verlag, 1998.
13. H. Seidl and C. Fecht. Interprocedural Analyses: A Comparison. *Journal of Logic Programming (JLP)*, 43(2):123–156, 2000.

A Proof of Proposition 2

We only prove statement (3). Let M_1, M_2 be non-empty subsets of E^*. By statement (1), we have

$$\begin{aligned}[M_1] \otimes [M_2] &= [M_1] \circ [M_2] \sqcup [M_2] \circ [M_1] \\ &= [M_2 \cdot M_1 \cup M_1 \cdot M_2] \qquad \sqsubseteq [M_1 \otimes M_2]\end{aligned}$$

Therefore, it remains to prove the reverse inequality. For that consider $w = e_1 \ldots e_m \in M_1 \otimes M_2$ where for disjoint index sets I_1, I_2 with $I_1 \cup I_2 = \{1, \ldots, m\}$, $w_i = w|_{I_i} \in M_i$. We claim:

$$[w] \sqsubseteq [w_1] \circ [w_2] \sqcup [w_2] \circ [w_1]$$

Clearly, this claim implies the statement (3) of our proposition. In order to prove the claim, let $[e_i] = \lambda x.(a_i \sqcap x) \sqcup b_i$, $i = 1, \ldots, m$, $[w_j] = \lambda x.(A_j \sqcap x) \sqcup B_j$, $j = 1, 2$, and $[w] = \lambda x.(A \sqcap x) \sqcup B$. Then by definition,

$$A = a_1 \sqcap \ldots \sqcap a_m = A_1 \sqcap A_2$$

Now consider value B. By definition,

$$B = \bigsqcup_{k=1}^{m} (b_k \sqcap a_{k+1} \sqcap \ldots \sqcap a_m)$$

We will show that for every k,

$$b_k \sqcap a_{k+1} \sqcap \ldots \sqcap a_m \sqsubseteq B_1 \sqcup B_2$$

W.l.o.g. assume that $k \in I_1$ (the case where $k \in I_2$ is completely analogous) and let $\{j_1, \ldots, j_r\} = \{j \in I_1 \mid j > k\}$. Then

$$b_k \sqcap a_{k+1} \sqcap \ldots \sqcap a_m \sqsubseteq b_k \sqcap a_{j_1} \sqcap \ldots \sqcap a_{j_r} \sqsubseteq B_1$$

which implies the assertion. □

B Proof of Theorem 2

Let us start with the following simple but useful observation:

Proposition 5. *For every $f \in \mathbb{F}$, $b \in \mathbb{D}$ and $\Delta = \lambda x.x \sqcup b$,*

$$f \otimes \Delta = \Delta \circ f$$

□

Next, we reformulate the constraint system for reachability as follows. We introduce the new values $[\![v]\!]'$, v a program point, and $[\![p]\!]'$, p a procedure, which collect the least upper bounds of *directly* reaching values by ignoring possible interleavings with execution paths possibly executed in parallel to v (or p). These values are determined as the least solution of the following constraint system:

$$\begin{array}{lll} [\![\mathsf{main}]\!]' \sqsupseteq d_0 & & (1) \\ [\![v]\!]' \sqsupseteq [v] \circ [\![p]\!]' & v \text{ program point in procedure } p & (2) \\ [\![p]\!]' \sqsupseteq [\![u]\!]' & e = (u, \_) \text{ calls } p \text{ or } p \,||\, \_ \text{ or } \_ \,||\, p & (3) \end{array}$$

By standard fixpoint induction we find:

Proposition 6. *For all program points v and procedures p,*

$$[\![v]\!] = [\![v]\!]' \sqcup B(v) \qquad \textit{and} \qquad [\![p]\!] = [\![p]\!]' \sqcup B(p)$$

□

In order to understand the "nature" of the values $B(v)$, we consider the sets $P(v)$ of edges possibly executed in parallel with program points v. They are determined through the least solution of the following constraint system:

$$\begin{array}{ll} E(p) \supseteq \{e\} & e \text{ basic edge in procedure } p \\ E(p) \supseteq E(q) & \text{procedure } p \text{ calls } q \text{ or } q \,||\, \_ \text{ or } \_ \,||\, q \\ P(v) \supseteq P(p) & v \text{ program point in } p \\ P(p) \supseteq P(u) & (u, \_) \text{ calls procedure } p \\ P(q_i) \supseteq E(q_{3-i}) \cup P(u) & (u, \_) \text{ calls } q_1 \,||\, q_2 \end{array}$$

By comparison of this constraint system with the definition of the values $\sigma(p)$ and $B(v), B(p)$ in section 6, we obtain:

Proposition 7. *For every procedure p and program point v,*

1. $\lambda x.x \sqcup \sigma(p) = I \sqcup [E(p)]$;
2. $\lambda x.x \sqcup B(p) = I \sqcup [P(p)]$ *and* $\lambda x.x \sqcup B(v) = I \sqcup [P(v)]$. □

Moreover, we have:

Proposition 8. *For every procedure p,*

$$[pre(\Pi(p))] = I \sqcup \bigsqcup [E(p)] = \lambda x.x \sqcup \sigma(p)$$ □

In order to simplify the proof of theorem 2, let us assume that all calls are parallel calls $q_1 \,||\, q_2$. This assumption does not incur a restriction, since an ordinary call to a procedure p can easily be simulated by a call to $p \,||\, q_0$ where q_0 is a procedure with just a single program point and no edges at all. Furthermore, it suffices to prove the assertion of the theorem just for program points v (the assertion for procedures then is an immediate consequence). We want to prove that for every program point v, the value $[\![v]\!]$ is a safe approximation of the value $\mathsf{Reach}(v)$, i.e., $[\![v]\!] \sqsupseteq \mathsf{Reach}(v)$. By definition,

$$\mathsf{Reach}(v) = [\Pi_r(v)]\, d_0 = [\Pi(v, \mathsf{main})]\, d_0$$

Therefore, let $w \in \Pi(v, \mathsf{main})$. Then there are program points $u_0, \ldots, u_m$, execution paths $w_0, \ldots, w_m$ together with execution paths w_i', procedures $q_1^{(i)}, q_2^{(i)}$ and indices $j(i) \in \{1, 2\}$ for $i = 1, \ldots, m$ such that:

- $u_m = v$;
- $w_i \in \Pi(u_i)$ for $i = 0, \ldots, m$;
- there are calls $(u_{i-1}, \_)$ to $q_1^{(i)} \,||\, q_2^{(i)}$;
- u_0 is a program point in main and for $i > 0$, u_i is a program point in $q_{j(i)}^{(i)}$;
- $w_i' \in \mathrm{pre}(\Pi(q_{3-j(i)}^{(i)}))$ for $i = 1, \ldots, m$;
- $w \in \{w_0\} \cdot (\{w_1'\} \otimes (\{w_1\} \cdot (\ldots \{w_{m-1}'\} \otimes (\{w_{m-1}\} \cdot (\{w_m'\} \otimes \{w_m\}))\ldots)))$.

Let $\Delta = \lambda x.x \sqcup P(v)$. Then by proposition 8,

$$[w_i'] \sqsubseteq [\mathrm{pre}(\Pi(q_{3-j(i)}^{(i)}))] = I \sqcup [E(q_{3-j(i)}^{(i)})] \sqsubseteq I \sqcup [P(v)] = \Delta$$

for all $i = 1, \ldots, m$. Therefore by proposition 5,

$$\begin{aligned}
[w] &\sqsubseteq (((\ldots(([w_m] \otimes [w_m']) \circ [w_{m-1}]) \otimes [w_{m-1}'] \ldots) \circ [w_1]) \otimes [w_1']) \circ [w_0] \\
&\sqsubseteq (((\ldots(([w_m] \otimes \Delta) \circ [w_{m-1}]) \otimes \Delta \ldots) \circ [w_1]) \otimes \Delta) \circ [w_0] \\
&= \Delta \circ (\ldots(([w_m] \otimes \Delta) \circ [w_{m-1}]) \otimes \Delta \ldots) \circ [w_1] \circ [w_0] \\
&\ldots \\
&= \Delta \circ [w_m] \circ [w_{m-1}] \circ \ldots \circ [w_0]
\end{aligned}$$

Since $([w_m] \circ \ldots \circ [w_0])\, d_0 \sqsubseteq [\![v]\!]'$, we conclude that

$$[w]\, d_0 \sqsubseteq \Delta\, [\![v]\!]' = [\![v]\!]' \sqcup B(v) = [\![v]\!]$$

which we wanted to prove.
It remains to prove the reverse inequality, i.e., that (1) $[\![v]\!]' \sqsubseteq \mathsf{Reach}(v)$ and (2) $B(v) \sqsubseteq \mathsf{Reach}(v)$.
Let us first consider inequality (1). The value $[\![v]\!]'$ is the least upper bound on values $[w]\, d_0$ such that there exit program points $u_0, \ldots, u_m$, execution paths $w_0, \ldots, w_m$ together with procedures $q_1^{(i)}, q_2^{(i)}$ and indices $j(i) \in \{1, 2\}$ for $i = 1, \ldots, m$ such that:

- $u_m = v$;
- $w_i \in \Pi(u_i)$ for $i = 0, \ldots, m$;
- there are calls $(u_{i-1}, \_)$ to $q_1^{(i)} \,||\, q_2^{(i)}$;
- u_0 is a program point in main and for $i > 0$, u_i is a program point in $q_{j(i)}^{(i)}$;
- $w = w_0 \ldots w_m$.

By induction on $r = m - i$ (from $r = 0$ to $r = m - 1$), we find that for $i > 0$,

$$w_i \ldots w_m \in \Pi(v, q_{j(i)}^{(i)})$$

and for $i = 0$,

$$w = w_0 \ldots w_m \in \Pi(v, \mathsf{main}) = \Pi_r(v)$$

Therefore,

$$[w]\, d_0 \sqsubseteq [\Pi_r(v)]\, d_0 = \mathsf{Reach}(v)$$

which we wanted to prove.
Now let us consider inequality (2). By proposition 7, $\lambda x.x \sqcup B(v) = I \sqcup [P(v)]$. Therefore, it suffices to prove for each edge $e \in P(v)$, that $b_e \sqsubseteq \mathsf{Reach}(v)$.
Since $e \in P(v)$, there exist program points $u_0, \ldots, u_m$, execution paths $w_0, \ldots, w_m$ together with procedures $q_1^{(i)}, q_2^{(i)}$, indices $j(i) \in \{1, 2\}$ for $i = 1, \ldots, m$, an index $k \in \{1, \ldots, m\}$ and one execution path w' such that

- $u_m = v$;
- $w_i \in \Pi(u_i)$ for $i = 0, \ldots, m$;
- there are calls $(u_{i-1}, \_)$ to $q_1^{(i)} \,||\, q_2^{(i)}$;
- u_0 is a program point in main and for $i > 0$, u_i is a program point in $q_{j(i)}^{(i)}$;
- $w'e \in \mathrm{pre}(\Pi(q_{3-j(k)}^{(k)})$.

As above, we conclude that $w_k \ldots w_m \in \Pi(v, q_{j(k)}^{(k)})$. By definition, then also

$$w_{k-1} w_k \ldots w_m w' e \in \Pi(v, q_{j(k-1)}^{(k-1)})$$

(where in case $k = 1$, we let $q_{j(0)}^{(0)} = \mathsf{main}$) and therefore also

$$w_0 \ldots w_{k-1} w_k \ldots w_m w' e \in \Pi(v, \mathsf{main}) = \Pi_r(v)$$

We conclude that

$$b_e \sqsubseteq b_e \sqcup (a_e \sqcap ([w_0 \ldots w_m w']\ d_0)) = [w_0 \ldots w_m w' e]\, d_0 \sqsubseteq [\Pi_r(v)]\, d_0 = \mathsf{Reach}(v)$$

which completes the proof. $\square$