Precise Fixed Point Based Analysis of Programs with Thread-Creation

Peter Lammich and Markus Müller-Olm

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
peter.lammich@uni-muenster.de and mmo@math.uni-muenster.de

Abstract. This paper presents an efficient, fixed point based algorithm for precise kill/gen analysis of interprocedural flow graphs with thread creation. The main idea of the algorithm is to separate a path reaching a control node into those steps required to reach the node and interfering steps, that are executed concurrently. These two parts are analyzed separately and combined afterwards. Exploiting the structure of kill/gen analysis we can show soundness and precision.

Key words: Interprocedural program analysis, parallelism, fixed-point based program analysis, thread-creation, bitvector problems, kill/gen problems

1 Introduction

As programming languages with explicit support for parallelism, such as Java, have become popular, the interest in analysis of parallel programs has increased in recent years. Most papers on precise analysis, such as [5, 11, 7, 9, 8, 3, 4], use fork/join as a model for parallelism. However, this is not adequate for analyzing languages like Java, because in presence of procedures or methods the thread-creation primitives used in such languages cannot be simulated by fork-join [1]. This paper presents a fixed-point based approach to precise kill/gen analysis of programs with thread-creation.

For parallel programs with (potentially recursive) procedures and synchronization statements, already reachability analysis is undecidable [10], even if data is ignored and guarded branching is abstracted by nondeterministic branching, as common in program analysis. So this paper, as well as other related work like [11, 1], ignores synchronization statements completely.

In this paper, we concentrate on kill/gen problems, a simple but yet practically relevant class of dataflow problems that comprise the more well-known bitvector problems (e.g. live variables, available expressions, etc.) Note that only slightly more powerful analyses, like copy constants or truly life variables are intractable or even undecidable (depending on the atomicity of assignments) for parallel programs [7, 9, 8], even when synchronization is ignored.

For programs with fork/join parallelism, Knoop, Steffen and Vollmer [5] have proposed an efficient fixed-point based approach for precise intraprocedural analysis of bitvector problems. This approach has been generalized to an interprocedural analysis of kill/gen-problems by Seidl and Steffen [11]. For programs with thread-creation, Bouajjani et al. [1] have developed an efficient automata theoretic approach to precise interprocedural bitvector analysis. Using tree automata, they have also generalized their approach to support some weak form of synchronization capable of simulating fork-join. However, to the best of our knowledge there has been no fixed-point based approach to precise interprocedural analysis of programs with thread creation.

The aim of this paper is to close this gap by developing an efficient fixed-point based approach for precise interprocedural kill/gen analysis of programs with thread-creation. We adopt the algorithmic idea of [5, 11] and equip it with a new analysis of possible interference for programs with thread-creation. Our approach can also be generalized to programs with both fork/join and thread creation, as described in the first author's diploma thesis [6].

This paper is organized as follows: In Section 2 we define parallel flow graphs used as program representation and equip them with an operational semantics. In Section 3 we define kill/gen analyses and the MOP-solution, that we want to compute precisely or approximately. In Section 4 we characterize the MOP-solution by means of *directly reaching paths* and *possible interference*, following an idea from [11]. In Section 5, we develop constraint systems for the directly reaching paths and possible interference and in Section 6, we use abstract interpretation [2] to transform these constraint systems into those with efficiently computable least solutions, that we can use to correctly approximate the MOP-solution. For the class of *positive distributive* kill/gen analyses, that comprises many practically relevant analyses, in particular all bitvector analyses, this approximation is even precise. Finally, in Section 7, we give a short conclusion and discuss future research.

2 Parallel Flow Graphs

A parallel flowgraph $(P, (G_p)_{p \in P})$ consists of a finite set P of procedure names, with $\mathsf{main} \in P$. For each procedure $p \in P$, there is a directed, edge annotated finite graph $G_p = (N_p, E_p, \mathsf{e}_p, \mathsf{r}_p)$ where N_p is the set of control nodes of procedure p and $E_p \subseteq N_p \times \mathcal{A} \times N_p$ is the set of edges that are annotated with base, call or spawn statements: $\mathcal{A} ::= \mathsf{base}\ b \mid \mathsf{call}\ p \mid \mathsf{spawn}\ p$ where b ranges over $\mathcal{B}$ and p over P. The set $\mathcal{B}$ of base edge annotations depends on the program being analyzed and is not specified further for the rest of this paper. $\mathsf{e}_p, \mathsf{r}_p \in N_p$ denote the entry and return node of a procedure $p \in P$. As usual we assume that the nodes of the procedures are disjoint, i.e. $N_p \cap N_{p'} = \emptyset$ for $p \neq p'$ and define $N = \bigcup_{p \in P} N_p$ and $E = \bigcup_{p \in P} E_p$.

We use $\mathcal{M}(X)$ to denote the set of multisets of elements from X, $\emptyset$ for the empty multiset, $\{a\}$ for the multiset containing the element a once and $A \uplus B$ for multiset union. Then we describe the operational semantics of a flowgraph by a labeled transition system $\overset{\cdot}{\longrightarrow} \subseteq \mathsf{Conf} \times \mathcal{L} \times \mathsf{Conf}$ over configurations $\mathsf{Conf} := \mathcal{M}(N^*)$ and labels $\mathcal{L} := E \cup \{\mathsf{ret}\}$. A configuration consists of the stacks of all threads running in parallel. A stack is modeled as a list of control nodes, the first element being the current control node. Each transition is labeled with the corresponding edge in the flowgraph or with ret for a procedure return. We use an *interleaving semantics*, nondeterministically picking the thread to make a transition among all available threads. Thus, we define $\overset{\cdot}{\longrightarrow}$ by the following inference rules, that describe the desired behavior:

[base]	$(\{[u]r\} \uplus c)\xrightarrow{e}(\{[v]r\} \uplus c)$	for edges $e = (u, \mathsf{base}\ a, v) \in E$
[call]	$(\{[u]r\} \uplus c)\xrightarrow{e}(\{[\mathsf{e}_q][v]r\} \uplus c)$	for edges $e = (u, \mathsf{call}\ q, v) \in E$
[ret]	$(\{[\mathsf{r}_q]r\} \uplus c)\xrightarrow{\mathsf{ret}}(\{r\} \uplus c)$	for procedures $q \in P$
[spawn]	$(\{[u]r\} \uplus c)\xrightarrow{e}(\{[v]r\} \uplus \{[\mathsf{e}_q]\} \uplus c)$	for edges $e = (u, \mathsf{spawn}\ q, v) \in E$

3 Dataflow Analysis

Dataflow analysis provides a generic, lattice based framework for constructing program analyses. A specific analysis is specified by a tuple $(L, \sqsubseteq, x_0, f)$ where $(L, \sqsubseteq)$ is a complete lattice representing analysis information, $x_0 \in L$ is the initial analysis information and $f : \mathcal{L} \to (L \overset{\mathsf{mon}}{\to} L)$ maps transition labels e to monotonic functions f_e that describe how a transition labeled e transforms analysis information. We assume that only base-transitions have transformers other than the identity.

In this paper we consider kill/gen-analyses, i.e. we require $(L, \sqsubseteq)$ to be distributive and the transformers to have the form $f_e(x) = (x \sqcap a_e) \sqcup b_e$ for some $a_e, b_e \in L$. Note that all transformers of this form are monotonic and that the set of these transformers is closed under composition of functions. Kill/gen-analysis comprise classic analyses like determination of available expressions or potentially uninitialized variables.

We are interested in the forward MOP-solution of a dataflow analysis $(L, \sqsubseteq, x_0, f)$ and a flowgraph (P, G), that is defined for each node $u \in N$ as:

$$\mathsf{MOP}[u] := \bigsqcup_{w \in \mathsf{Reach}[u]} f_w(x_0)$$

where $\mathsf{Reach}[u] := \{w \mid \exists r, c' : \{[\mathsf{e}_{\mathsf{main}}]\}\xrightarrow{w}\{[u]r\} \uplus c'\}$ is the set of paths reaching node u from the initial configuration $\{[\mathsf{e}_{\mathsf{main}}]\}$, $f_{e_1,\ldots,e_n} := f_{e_n} \circ \ldots \circ f_{e_1}$ with $f_\varepsilon := \lambda x.x$, and $\xrightarrow{w}$ is the natural extension of $\overset{\cdot}{\longrightarrow}$ to sequences of labels. $\mathsf{MOP}[u]$ is the smallest, i.e. most precise analysis information valid after all executions reaching control node u.

4 Possible Interference

It follows from results in [1] that the sets $\mathsf{Reach}[u]$ cannot be characterized as least solution of a system of equations or inequations on sets of paths (constraint systems) using the natural operators ,,concatenation" and ,,interleaving" from [11]. Therefore we cannot compute the MOP-solution directly by an abstract interpretation of such a constraint system.

To avoid this problem, in this section we derive an alternative characterization of the MOP-solution as the join of two values, each of which can be computed by abstract interpretation. The idea is to split a reaching path into *directly reaching transitions* and *interfering transitions*, as illustrated by Figure 1. The vertical lines are the executions of the threads and the horizontal arrows symbolize thread creation. The execution depicted in this figure reaches, possibly among others, the control node u. The directly reaching transitions w.r.t. to u are marked with thick lines. The other transitions are interfering transitions, that are executed concurrently with the directly reaching transitions. The path depicted by this figure is some interleaving of the directly reaching and the interfering transitions.

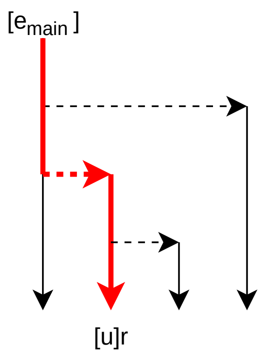

Fig. 1: Splitting a reaching path.

To formalize the directly reaching paths, we define the *direct transition relation* $\overset{\cdot}{\Longrightarrow} \subseteq (N^* \times \mathcal{M}(N^*)) \times \mathcal{L} \times (N^* \times \mathcal{M}(N^*))$, by the following inference rules:

[base]	$([u]r, c) \overset{e}{\Longrightarrow} ([v]r, c)$	for edges $e = (u, \mathsf{base}\ a, v) \in E$
[call]	$([u]r, c) \overset{e}{\Longrightarrow} ([\mathsf{e}_q][v]r, c)$	for edges $e = (u, \mathsf{call}\ q, v) \in E$
[ret]	$([\mathsf{r}_q]r, c) \overset{\mathsf{ret}}{\Longrightarrow} (r, c)$	for procedure $q \in P$
[spawn1]	$([u]r, c) \overset{e}{\Longrightarrow} ([v]r, \{[\mathsf{e}_q]\} \uplus c)$	for edges $e = (u, \mathsf{spawn}\ q, v) \in E$
[spawn2]	$([u]r, c) \overset{e}{\Longrightarrow} ([\mathsf{e}_q], \{[v]r\} \uplus c)$	for edges $e = (u, \mathsf{spawn}\ q, v) \in E$

Intuitively, the first component of the state is the current thread's stack and the second component collects all stacks that can make interfering transitions. The rules are similar to that of $\overset{\cdot}{\longrightarrow}$, but only can operate on the current thread's stack. On a spawn transition, either the spawning thread remains the current thread ([spawn1]), or the spawned thread becomes the current thread ([spawn2]).

A key observation is that due to the lack of synchronization, the interfering transitions are mostly independent from the directly reaching transitions. The only dependence is, that the thread executing an interfering transition must have been already created. When given a reaching path to a control node u, we can move the interfering transitions to the end of that path, thus getting a path with a prefix of directly reaching transitions and a suffix of interfering transitions, that reaches the *same* configuration as the original path. Thereby the interfering transitions will not affect the thread that reached u any more.

Accordingly, we define the set of directly reaching paths to u as

$$\mathsf{R}^{\mathsf{op}}[u] := \{w \mid \exists r, \hat{c} : ([\mathsf{e}_{\mathsf{main}}], \emptyset) \overset{w}{\Longrightarrow} ([u]r, \hat{c})\}$$

and the possible interference at u as

$$\mathsf{PI}^{\mathsf{op}}[u] := \{e \mid \exists r, \hat{c}, c', w : ([\mathsf{e}_{\mathsf{main}}], \emptyset) \overset{*}{\Longrightarrow} ([u]r, \hat{c}) \wedge \hat{c} \xrightarrow{w[e]} c'\}$$

Intuitively, $\mathsf{R}^{\mathsf{op}}[u]$ captures the set of *directly* reaching paths to u, and $\mathsf{PI}^{\mathsf{op}}[u]$ captures the set of interfering transitions that may be executed on some reaching path to u. In contrast to Reach, both R^{op} and $\mathsf{PI}^{\mathsf{op}}$ can be characterized by constraint systems as we will see in the next section.

Now we can prove the following theorem:

Theorem 1. *For kill/gen-analyses, we have at each control node $u \in N$:*

$$\mathsf{MOP}[u] = \alpha_{\mathsf{R}}(\mathsf{R}^{\mathsf{op}}[u]) \sqcup \alpha_{\mathsf{PI}}(\mathsf{PI}^{\mathsf{op}}[u])$$

with $\alpha_{\mathsf{R}}(W) := \bigsqcup\{f_w(x_0) \mid w \in W\}$ *and* $\alpha_{\mathsf{PI}}(E) := \bigsqcup\{b_e | e \in E\}$.

Proof. (Sketch): For the $\sqsubseteq$-direction, we observe that the transitions on a reaching path to u are either directly reaching transitions, and thus contained in $\mathsf{R}^{\mathsf{op}}[u]$ or interfering transitions, that could also be executed at the end of the reaching path and are thus contained in $\mathsf{PI}^{\mathsf{op}}[u]$. The proposition then follows from a structural property of kill/gen problems also exploited in [11], that states that for $w \in w_1 \otimes w_2$ we have $f_w(x) \sqsubseteq f_{w_1}(x) \sqcup \bigsqcup\{b_e \mid e \in w_2\}$ for any kill/gen function f and argument x.

For the $\sqsupseteq$-direction, we first observe that any directly reaching path is also a reaching path, and hence $\mathsf{MOP}[u] \sqsupseteq \alpha_{\mathsf{R}}(\mathsf{R}^{\mathsf{op}}[u])$.

For each transition $e \in \mathsf{PI}^{\mathsf{op}}[u]$, we obviously can construct a reaching path to u, that executes e as last transition. Let this path be $w[e]$. The transfer function of $w[e]$ has the form $\lambda x.f_w(x) \sqcap a_e \sqcup b_e$, and is thus greater than b_e for any argument, and hence we have $\mathsf{MOP}[u] \sqsupseteq \alpha_{\mathsf{PI}}(\mathsf{PI}^{\mathsf{op}}[u])$ and altogether the proposition follows. □

5 Constraint Systems

In order to compute the MOP-solution, we will characterize R^{op} and $\mathsf{PI}^{\mathsf{op}}$ as the least solutions $\mathsf{lfp}(\mathsf{R})$ and $\mathsf{lfp}(\mathsf{PI})$ of constraint systems R and PI. In Section 6 we then use abstract interpretation to compute $\alpha_{\mathsf{R}}(\mathsf{lfp}(\mathsf{R}))$ and $\alpha_{\mathsf{PI}}(\mathsf{lfp}(\mathsf{PI}))$ by fixpoint iteration.

For technical reasons, we assume that every edge $e \in E$ is *dynamically reachable*, i.e. that $\mathsf{Reach}[u] \neq \emptyset$ if u has an outgoing edge. For general flowgraphs, we can detect the unreachable edges by a simple analysis and then remove them. This obviously does not affect Reach or MOP.

In order to treat procedures when characterizing R^{op} we adopt a standard technique from interprocedural analysis, that first regards so called *same-level paths* captured in a constraint system S over variables $\mathsf{S}[u], u \in N$ and then uses these to assemble all reaching paths in the constraint system R over variables $\mathsf{R}[u], u \in N$:

[inits]	$\mathsf{S}[\mathsf{e}_q] \supseteq \{\varepsilon\}$	for $q \in P$
[base]	$\mathsf{S}[v] \supseteq \mathsf{S}[u]; \{[e]\}$	for $e = (u, \mathsf{base}\ \_, v) \in E$
[call]	$\mathsf{S}[v] \supseteq \mathsf{S}[u]; \{[e]\}; \mathsf{S}[\mathsf{r}_q]; \{[\mathsf{ret}]\}$	for $e = (u, \mathsf{call}\ q, v) \in E$
[spawn]	$\mathsf{S}[v] \supseteq \mathsf{S}[u]; \{[e]\}$	for $e = (u, \mathsf{spawn}\ q, v) \in E$

[init]	$\mathsf{R}[\mathsf{e}_{\mathsf{main}}] \supseteq \{\varepsilon\}$	
[reach]	$\mathsf{R}[u] \supseteq \mathsf{R}[\mathsf{e}_p]; \mathsf{S}[u]$	for $u \in N_p$
[callt]	$\mathsf{R}[\mathsf{e}_q] \supseteq \mathsf{R}[u]; \{[e]\}$	for $e = (u, \mathsf{call}\ q, \_) \in E$
[spawnt]	$\mathsf{R}[\mathsf{e}_q] \supseteq \mathsf{R}[u]; \{[e]\}$	for $e = (u, \mathsf{spawn}\ q, \_) \in E$

The operator ; denotes list concatenation, lifted to sets of lists. The main novelty here is the constraint [spawn], that allows a same-level path to contain spawn edges, and the constraint [spawnt], that coincides with the [spawn2]-rule from the definition of $\overset{\cdot}{\Longrightarrow}$ and allows a reaching path to 'switch' to the spawned thread.

With $\mathsf{lfp}(\mathsf{X})$ we denote the least solution of a constraint system X, if it exists. The existence of $\mathsf{lfp}(\mathsf{S})$ and $\mathsf{lfp}(\mathsf{R})$ follows from the Knaster-Tarski fixed point theorem and, using standard techniques from program analysis, we can show $\mathsf{lfp}(\mathsf{R}) = \mathsf{R}^{\mathsf{op}}$.

$\mathsf{PI}^{\mathsf{op}}$ can be characterized by the following constraint system, that is split into four parts for the sake of clarity:

[GP.init]	$\mathsf{GP}[u] \supseteq \{e\}$	for $e = (u, \_, \_) \in E$
[GP.edge]	$\mathsf{GP}[u] \supseteq \mathsf{GP}[v]$	for $(u, \mathsf{base}\ \_, v) \in E$ or $(u, \mathsf{spawn}\ \_, v) \in E$
[GP.calle]	$\mathsf{GP}[u] \supseteq \mathsf{GP}[v]$	for $(u, \mathsf{call}\ q, v) \in E$ if q can terminate
[GP.trans]	$\mathsf{GP}[u] \supseteq \mathsf{GP}[\mathsf{e}_q]$	for $(u, \mathsf{call}\ q, v) \in E$ or $(u, \mathsf{spawn}\ q, v) \in E$

[SP.edge]	$\mathsf{SP}[u] \supseteq \mathsf{SP}[v]$	for $(u, \mathsf{base}\ \_, v) \in E$ or $(u, \mathsf{spawn}\ \_, v) \in E$
[SP.calle]	$\mathsf{SP}[u] \supseteq \mathsf{SP}[v]$	for $(u, \mathsf{call}\ q, v) \in E$ if q can terminate
[SP.callt]	$\mathsf{SP}[u] \supseteq \mathsf{SP}[\mathsf{e}_q]$	for $(u, \mathsf{call}\ q, v) \in E$ if v can terminate
[SP.spawnt]	$\mathsf{SP}[u] \supseteq \mathsf{GP}[\mathsf{e}_q]$	for $(u, \mathsf{spawn}\ q, v) \in E$ if v can terminate

[DG.init]	$\mathsf{DG}[u] \supseteq \{e\}$	for $e = (u, \_, \_) \in E$
[DG.edge]	$\mathsf{DG}[u] \supseteq \mathsf{DG}[v]$	for $(u, \mathsf{base}\ \_, v) \in E$ or $(u, \mathsf{spawn}\ \_, v) \in E$
[DG.calle]	$\mathsf{DG}[u] \supseteq \mathsf{DG}[v]$	for $(u, \mathsf{call}\ q, v) \in E$ if q can terminate
[DG.trans]	$\mathsf{DG}[u] \supseteq \mathsf{DG}[\mathsf{e}_q]$	for $(u, \mathsf{call}\ q, v) \in E$ or $(u, \mathsf{spawn}\ q, v) \in E$
[DG.deepen]	$\mathsf{DG}[\mathsf{r}_q] \supseteq \mathsf{DG}[v]$	for $(u, \mathsf{call}\ q, v) \in E$ if q can terminate

[PI.edge]	$\mathsf{PI}[v] \supseteq \mathsf{PI}[u]$	for $(u, \mathsf{base}\ \_, v) \in E$ or $(u, \mathsf{spawn}\ \_, v) \in E$
[PI.calle]	$\mathsf{PI}[v] \supseteq \mathsf{PI}[u]$	for $(u, \mathsf{call}\ q, v) \in E$ if q can terminate
[PI.trans]	$\mathsf{PI}[\mathsf{e}_q] \supseteq \mathsf{PI}[u]$	for $(u, \mathsf{call}\ q, v) \in E$ or $(u, \mathsf{spawn}\ q, v) \in E$
[PI.calli]	$\mathsf{PI}[v] \supseteq \mathsf{SP}[\mathsf{e}_q]$	for $(u, \mathsf{call}\ q, v) \in E$
[PI.spawn1]	$\mathsf{PI}[v] \supseteq \mathsf{GP}[\mathsf{e}_q]$	for $(u, \mathsf{spawn}\ q, v) \in E$
[PI.spawn2]	$\mathsf{PI}[\mathsf{e}_q] \supseteq \mathsf{DG}[v]$	for $(u, \mathsf{spawn}\ q, v) \in E$

Here we say that a procedure $p \in P$ can terminate, iff there exists a same-level path from e_p to r_p. Accordingly, a node $v \in N_p$ can terminate iff there exists a same-level path from v to r_p. This information can be determined by a simple abstract interpretation of the constraint system S for same-level paths.

The proof that $\mathsf{lfp}(\mathsf{PI}) = \mathsf{PI}^{\mathsf{op}}$ is omitted here due to lack of space, but we will try to give an intuition about the constraints. The constraint [PI.edge] propagates the possible interference along a base edge, [PI.calle] along a call edge and [PI.trans] into a called or spawned procedure. [PI.calli] adds the possible interference caused by threads that are created during a run through the called procedure to the end node of the call edge. This interference is called the *spawn potential* of the procedure and characterized by the SP-part of the constraint system. [PI.spawn1] accounts for the interference caused by the created thread in the creator thread and [PI.spawn2] accounts for the interference caused by the creator thread in the created thread. The constraints for the former *generate potential* (GP) and the latter *deep generate potential* (DG) differ only in the [DG.deepen] constraint that has no correspondence for GP. This constraint captures that the creator thread may have a non-empty return stack, so that also transitions of procedures deeper on the stack can cause interference to the created thread.

6 Algorithm

It remains to compute $\alpha_{\mathsf{R}}(\mathsf{lfp}(\mathsf{R}))$ and $\alpha_{\mathsf{PI}}(\mathsf{lfp}(\mathsf{PI}))$. For this purpose we replace the operators on path sets in R and PI in a standard way by operators on L, obtaining constraint systems $\mathsf{R}^{\#}$ and $\mathsf{PI}^{\#}$ over L. By standard results from abstract interpretation [2], we get

Theorem 2. *In general, we have* $\alpha_{\mathsf{PI}}(\mathsf{lfp}(\mathsf{PI})) = \mathsf{lfp}(\mathsf{PI}^{\#})$ *and* $\alpha_{\mathsf{R}}(\mathsf{lfp}(\mathsf{R})) \sqsubseteq \mathsf{lfp}(\mathsf{R}^{\#})$. *For positive distributive transfer functions, we even have* $\alpha_{\mathsf{R}}(\mathsf{lfp}(\mathsf{R})) = \mathsf{lfp}(\mathsf{R}^{\#})$.

Now we can efficiently compute a safe and for positive distributive analysis even precise approximation of the MOP solution by the following algorithm:

1. Generate the constraint systems $\mathsf{R}^{\#}$ and $\mathsf{PI}^{\#}$, as well as abstract versions of S, DG, SP and GP, from the flowgraph.
2. Determine their least solutions.
3. Return the vector MFP with $\mathsf{MFP}[u] := \mathsf{R}^{\#}[u] \sqcup \mathsf{PI}^{\#}[u]$. The name MFP (*minimum fixpoint solution*) results from the characterization of the least solution of a constraint system as least fixed point of an induced function.

From Theorems 1, 2 and the soundness and precision of the constraint systems, we immediately get $\mathsf{MOP} \sqsubseteq \mathsf{MFP}$, and if the transformers are positive distributive, we even get $\mathsf{MOP} = \mathsf{MFP}$.

The constraint systems contain $O(|E|+|P|)$ constraints over $O(|N|)$ variables. If the height of $(L, \sqsubseteq)$ is bounded by $h(L)$ and a lattice operation (join, compare, assign) needs time $O(op)$, we can calculate MFP in time $O((|E| * h(L) + |N|) * op)$ using a worklist algorithm.

7 Conclusion

In this paper we have presented an efficient, fixed-point based algorithm for precise kill/gen analysis of interprocedural flowgraphs with thread creation. This model is closer to the one used in real programming languages such as Java or C++ than the fork/join model previously studied in the literature. In the first author's master thesis [6] all the proofs of a variant of the approach of this paper are elaborated.

Our work uses a similar idea as in [11], and indeed we can combine these two approaches resulting in a precise fixed-point based analysis of programs with both fork/join and thread creation. We implemented this combination in [6].

Our analysis does not consider synchronization. For programs with synchronization our analysis still is a correct (but poor) approximation. Precise analysis of interprocedural parallel flowgraphs with synchronization is undecidable in general [10], but further research needs to be done to increase approximation quality. Also extensions to more complex domains like dependence analysis, as studied in [8] for fork/join, have to be investigated.

Acknowledgements: We thank Helmut Seidl and Bernhard Steffen for interesting discussions on analyzing parallel programs.

References

1. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
3. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proc. of FoSSaCS'99*, pages 14–30. Springer, 1999.
4. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proc. of POPL'00*, pages 1–11. Springer, 2000.
5. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.
6. P. Lammich. Fixpunkt-basierte optimale Analyse von Programmen mit Thread-Erzeugung. Master's thesis, University of Dortmund, May 2006.
7. M. Müller-Olm. The complexity of copy constant detection in parallel programs. In *Proc. of STACS'01*, pages 490–501. Springer, 2001.
8. M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.
9. M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proc. of STOC'01*, pages 647–656, New York, NY, USA, 2001. ACM Press.
10. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.
11. H. Seidl and B. Steffen. Constrained-based inter-procedural analysis of parallel programs. In *Proc. of ESOP'00*. Springer, 2000.