The Complexity of Copy Constant Detection in Parallel Programs

Markus Müller-Olm

Universität Dortmund, FB Informatik, LS V,
44221 Dortmund, Germany
mmo@ls5.cs.uni-dortmund.de

Abstract. Despite of the well-known state-explosion problem, certain simple but important data-flow analysis problems known as gen/kill problems can be solved efficiently and completely for parallel programs with a shared state [7, 6, 2, 3, 13]. This paper shows that, in all probability, these surprising results cannot be generalized to significantly larger classes of data-flow analysis problems.
More specifically, we study the complexity of detecting copy constants in parallel programs, a problem that may be seen as representing the next level of difficulty of data-flow problems beyond gen/kill problems. We show that already the intraprocedural problem for loop-free parallel programs is co-NP-complete and that the interprocedural problem is even PSPACE-hard.

1 Introduction

A well-known obstacle for the automatic analysis of parallel programs is the so-called *state-explosion problem*: the number of (control) states of a parallel program grows exponentially with the number of parallel components. It comes as a surprise that certain basic but important data-flow analysis problems can nevertheless be solved completely and efficiently for programs with a fork/join kind of parallelism.

Knoop, Steffen, and Vollmer [7] show that *bitvector analyses*, which comprise, e.g., live/dead variable analysis, available expression analysis, and reaching definition analysis [8], can efficiently be performed on such programs. Knoop shows in [6] that a simple variant of constant detection, that of so-called *strong constants*, is tractable as well. These articles restrict attention to the *intraprocedural* problem, in which each procedure body is analyzed separately with worst-case assumption on called procedures. Seidl and Steffen [13] generalize these results to the *interprocedural* case in which the interplay between procedures is taken into account and to a slightly more extensive class of data-flow problems called *gen/kill problems*[1]. All these papers extend the fixpoint computation technique

[1] Gen/kill problems are characterized by the fact that all transfer functions are of the form $\lambda x.(x \wedge a) \vee b$, where a, b are constants from the underlying lattice of data-flow facts.

common in data-flow analysis to parallel programs. Another line of research applies automata-theoretic techniques that were originally developed for the verification of PA-processes, a certain class of infinite-state processes combining sequentiality and parallelism. Specifically, Esparza, Knoop, and Podelski [2, 3] demonstrate how live variables analysis can be done and indicate that other bitvector analyses can be approached in a similar fashion.

Can these results be generalized further to considerably richer classes of data-flow problems? The current paper shows that this is very unlikely. We investigate the complexity of detection of copy constants, a problem that may be seen as a canonic representative of the next level of difficulty of data-flow problems beyond gen/kill problems. In the sequential setting the problem gives rise to a *distributive* data-flow framework on a lattice with small chain height and can thus – by the classic result of Kildall [5, 8] – completely and efficiently be solved by a fixpoint computation. We show in this paper that copy constant detection is co-NP-complete already for loop-free parallel programs without procedures and becomes even PSPACE-hard if one allows loops and non-recursive procedures. This renders the possibility of complete and efficient data-flow analysis algorithms for parallel programs for more extensive classes of analyses unlikely, as it is generally believed that the inclusions P $\subseteq$ co-NP $\subseteq$ PSPACE are proper.

Our theorems should be contrasted with complexity and undecidability results of Taylor [14] and Ramalingam [11] who consider *synchronization-dependent* data-flow analyses of parallel programs, i.e. analyses that are precise with respect to the synchronization structure of programs. Taylor and Ramalingam largely exploit the strength of rendezvous style synchronization, while we exploit only interference and no kind of synchronization. Our results thus point to a more fundamental limitation in data-flow analysis of parallel programs.

This paper is organized as follows: In Sect. 2 we give some background information on data-flow analysis in general and the constant detection problem in particular. In Sect. 3 we introduce loop-free parallel programs. This sets the stage for the co-NP-completeness result for the loop-free intraprocedural parallel case which is proved afterwards. We proceed by enriching the considered programming language with loops and procedures in Sect. 4. We then show that the interprocedural parallel problem is PSPACE-hard even if we allow only non-recursive procedures. In the Conclusions, Sect. 5, we indicate that the presented results apply also to some other data-flow analysis problems, detection of may-constants and detection of faint code, and discuss directions for future research. Throughout this paper we assume that the reader is familiar with the basic notions and methods of the theory of computational complexity (see, e.g., [10]).

2 Copy Constants

The goal of *data-flow analysis* is to gather information about certain aspects of the behavior of programs by a static analysis. Such information is valuable e.g. in optimizing compilers and in CASE tools. However, most questions about programs are undecidable. This holds in particular for the question whether a condition in a program may be satisfied or not. In order to come to grips with undecidability, it is common in data-flow analysis to abstract from the conditions in the programs and to interpret conditional branching as non-deterministic branching, a point of view adopted in this paper. Of course, an analysis based on this abstraction considers more program executions than actually possible at run-time. One is careful to take this into account when exploiting the results of data-flow analysis.

An expression e is a *constant* at a given point p in a program, if e evaluates to one and the same value whenever control reaches p, i.e. after every run from the start of the program to p. If an expression is detected to be a constant at compile time it can be replaced by its value, a standard transformation in optimizing compilers known as *constant propagation* or *constant folding* [8]. Constant folding is profitable as it decreases both code size and execution time. Constancy information is sometimes also useful for eliminating branches of conditionals that cannot be taken at run-time and for improving the precision of other data-flow analyses.

Reif and Lewis [12] show by a reduction of Hilbert's tenth problem that the general constant detection problem in sequential programs is undecidable, even if branching is interpreted non-deterministically. However, if one restricts the kind of expressions allowed on the right hand side of assignment statements appropriately, the problem becomes decidable. (In practice assignments of a different form are treated by approximating or worst-case assumptions.) A problem that is particularly simple for sequential programs are so-called *copy constants.* In this problem assignment statements take only the simple forms $x := c$ (constant assignment) and $x := y$ (copying assignment), where c is a constant and x, y are variables. In the remainder of this paper we study the complexity of detecting copy constants in parallel programs.

3 Loop-Free Parallel Programs

Let us, first of all, set the stage for the parallel loop-free intraprocedural copy constant detection problem. We consider *loop-free parallel programs* given by the following abstract grammar,

$$\begin{aligned} \pi &::= x := e \mid \mathbf{write}(x) \mid \mathbf{skip} \mid \pi_1 \,;\, \pi_2 \mid \pi_1 \parallel \pi_2 \mid \pi_1 \sqcap \pi_2 \\ e &::= c \mid x\,, \end{aligned}$$

where x ranges over some set of variables and c over some set of basic constants. As usual we use parenthesis to disambiguate programs. Note that this language has only constant and copying assignments. The specific nature of basic constants and the value domain in which they are interpreted is immaterial; we only need that 0 and 1 are two constants representing different values, which – by abuse of notation – are also denoted by 0 and 1. The atomic statements of the language are assignment statements $x := e$ that assign the current value of e to variable x, 'do-nothing'-statements **skip**, and write-statements. The purpose of write-statements in this paper is to mark prominently the program points at which we are interested in constancy of a certain variable. The operator ; represents sequential composition, $\|$ represents parallel composition, and $\sqcap$ non-deterministic branching.

Parallelism is understood in an interleaving fashion; assignments and write-statements are assumed to be atomic. A run of a program is a maximal sequence of atomic statements that may be executed in this order in an execution of the program. The program $(x := 1 ; x := y) \parallel y := x$ for example, has the three runs $\langle x := 1, x := y, y := x\rangle$, $\langle x := 1, y := x, x := y\rangle$, and $\langle y := x, x := 1, x := y\rangle$.

In order to allow a formal definition of runs, we need some notation. We denote the empty sequence by ε and the concatenation operator by an infix dot. The concatenation operator is lifted to sets of sequences in the obvious way: If S, T are two sets of sequences then $S \cdot T = \{s \cdot t \mid s \in S, t \in T\}$. Let $r = \langle e_1, \dots, e_n\rangle$ be a sequence and $I = \{i_1, \dots, i_k\}$ a subset of positions in r such that $i_1 < i_2 < \dots < i_k$. Then $r|I$ is the sequence $\langle e_{i_1}, \dots, e_{i_k}\rangle$. We write $|r|$ for the length of r, viz. n. The *interleaving* of S and T is

$$S \parallel T \overset{\text{def}}{=} \{r \mid \exists I_S, I_T : I_S \cup I_T = \{1, \dots, |w|\}, I_S \cap I_T = \emptyset, r|I_S \in S, r|I_T \in T\}\,.$$

The set of runs of a program can now inductively be defined:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{Runs}(x := e) & = & \{\langle x := e\rangle\} & \mathsf{Runs}(\pi_1 \,;\, \pi_2) & = & \mathsf{Runs}(\pi_1) \cdot \mathsf{Runs}(\pi_2) \\
\mathsf{Runs}(\mathbf{write}(x)) & = & \{\langle \mathbf{write}(x)\rangle\} & \mathsf{Runs}(\pi_1 \parallel \pi_2) & = & \mathsf{Runs}(\pi_1) \parallel \mathsf{Runs}(\pi_2) \\
\mathsf{Runs}(\mathbf{skip}) & = & \{\varepsilon\} & \mathsf{Runs}(\pi_1 \sqcap \pi_2) & = & \mathsf{Runs}(\pi_1) \cup \mathsf{Runs}(\pi_2)\,.
\end{array}$$

3.1 NP-Completeness of the Loop-Free Intraprocedural Problem

The remainder of this section is devoted to the proof of the following theorem, which shows that complete detection of copy constants is intractable in parallel programs, unless P = NP.

Theorem 1. *The problem of detecting copy constants in loop-free parallel programs is co-NP-complete.*

Certainly, the problem lies in co-NP: if a variable x is *not* constant at a certain point in the program we can guess two runs that witness two different values. As the program has no loops, the length of these runs (and thus the time needed to guess them) is at most linear in the size of the program.

For showing co-NP-hardness we reduce SAT, the most widely known NP-complete problem [1, 10], to the negation of a copy constant detection problem. An instance of SAT is a conjunction $c_1 \wedge \dots \wedge c_k$ of *clauses* $c_1, \dots, c_k$. Each clause is a disjunction of *literals*; a literal l is either a variable x or a negated variable $\neg x$, where x ranges over some set of variables X. It is straightforward to define when a *truth assignment* $T : X \to \mathbb{B}$, where $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ is the set of truth values, satisfies $c_1 \wedge \dots \wedge c_k$. The SAT problem asks us to decide for each instance $c_1 \wedge \dots \wedge c_k$ whether there is a satisfying truth assignment or not.

Now suppose given a SAT instance $c_1 \wedge \ldots \wedge c_k$ with k clauses over n variables $X = \{x_1, \ldots, x_n\}$. We write $\bar{X} = \{\neg x_1, \ldots, \neg x_n\}$ for the set of negated variables. From this SAT instance we construct a loop-free parallel program. In the program we use $k+1$ variables $z_0, z_1, \ldots, z_k$. Intuitively, z_i is, for $1 \leq i \leq k$, related to clause c_i; z_0 is an extra variable.

For each literal $l \in X \cup \bar{X}$ we define a program π_l. Program π_l consists of a sequential composition of assignments of the form $z_i := z_{i-1}$ in increasing order of i. The assignment $z_i := z_{i-1}$ is in π_l if and only if the literal l makes clause i true. Formally, $\pi_l = \pi_l^k$, where

$$\pi_l^0 \overset{\text{def}}{=} \textbf{skip} \quad \text{and} \quad \pi_l^i \overset{\text{def}}{=} \begin{cases} \pi_l^{i-1} \ ; z_i := z_{i-1} \,, & \text{if clause } c_i \text{ contains } l \\ \pi_l^{i-1} \,, & \text{if clause } c_i \text{ does not contain } l \end{cases}$$

for $i = 1, \ldots, k$. Now, consider the following program π:

$$\begin{gathered} z_0 := 1 \ ; z_1 := 0 \ ; \ldots ; z_k := 0 \ ; \\ [(\pi_{x_1} \sqcap \pi_{\neg x_1}) \parallel \cdots \parallel (\pi_{x_n} \sqcap \pi_{\neg x_n})] \ ; \\ (z_k := 0 \sqcap \textbf{skip}) \ ; \textbf{write}(z_k) \,. \end{gathered}$$

Clearly, π can be constructed from the given SAT instance $c_1 \wedge \ldots \wedge c_k$ in polynomial time or logarithmic space. We show that the variable z_k at the write-statement is not a constant if and only if $c_1 \wedge \ldots \wedge c_k$ is satisfiable. This proves the co-NP-hardness claim.

First observe that 0 and 1 are the only values z_k can hold at the write-statement because all variables are initialized by 0 or 1 and the other assignments only copy these values. Clearly, due to the non-deterministic choice just before the write-statement, z_k may hold 0 finally. Thus, z_k is a constant at the write-statement iff it cannot hold 1 there. Hence, our goal reduces to proving that z_k can hold 1 finally if and only if $c_1 \wedge \ldots \wedge c_k$ is satisfiable.

"If": Suppose $T : X \to \mathbb{B}$ is a satisfying truth assignment for $c_1 \wedge \ldots \wedge c_k$. Consider the following run of π: in each parallel component $\pi_{x_i} \sqcap \pi_{\neg x_i}$ we choose the left branch π_{x_i} if $T(x_i) = \mathtt{tt}$ and the right branch $\pi_{\neg x_i}$ otherwise. As T is a satisfying truth assignment, there will be, for any $i \in \{1, \ldots, k\}$, at least one assignment $z_i := z_{i-1}$ in one of the chosen branches. We interleave the branches now in such a way that the assignment(s) to z_1 are executed first, followed by the assignment(s) to z_2 etc. This results in a run that copies the initialization value 1 of z_0 to z_k.

"Only if": Suppose z_k may hold 1 at the write-statement. As the initialization $z_0 := 1$ is the only statement in which the constant 1 occurs, there must be a run in which this value is copied from z_0 to z_k via a sequence of copy instructions. As all copying assignments in π have the form $z_i := z_{i-1}$, the value must be copied from z_0 to z_1, from z_1 to z_2 etc. Consequently, the non-deterministic choices in the parallel components can be resolved in such a way that the chosen branches contain all the assignments $z_i := z_{i-1}$ for $i = 1, \ldots, k$. From such a choice a satisfying truth assignment can easily be constructed.

4 Adding Loops and Procedures

Let us now consider a richer program class: programs with procedures, parallelism and loops. A *procedural parallel program* comprises a finite set Proc of *procedure names* containing a distinguished name *Main*. Each procedure name P is associated with a statement π_P, the corresponding *procedure body*, constructed according to the grammar

$$\begin{aligned} e &::= c \mid x \\ \pi &::= x := e \mid \textbf{write}(x) \mid \textbf{skip} \mid Q \mid \pi_1 \,;\, \pi_2 \mid \pi_1 \parallel \pi_2 \mid \pi_1 \sqcap \pi_2 \mid \pi^* \,, \end{aligned}$$

where Q ranges over Proc. A statement of the form Q represents a call to procedure Q and π^* stands for a loop that iterates π an indefinite number of times. Such an indefinite looping construct is consistent with the abstraction that branching is non-deterministic. A program is *non-recursive* if there is an order on the procedure names such that in the body of each procedure only procedures with a strictly smaller name are called.

The definition of runs from the previous section can easily be extended to the enriched language by the following two clauses:[2]

$$\mathsf{Runs}(\pi^*) = \mathsf{Runs}(\pi)^* \qquad\qquad \mathsf{Runs}(P) = \mathsf{Runs}(\pi_P)\,.$$

As usual, we define $X^* = \bigcup_{i\geq 0} X^i$, where $X^0 = \{\varepsilon\}$ and $X^{i+1} = X \cdot X^i$ for a set X of sequences. The runs of the program are the runs of *Main*.

4.1 PSPACE-Hardness of Interprocedural Copy Constant Detection

The goal of this section is to prove the following result.

Theorem 2. *The problem of detecting copy constants in non-recursive procedural parallel programs is PSPACE-hard.*

The proof is by means of a reduction of the QBF (quantified Boolean formulas) problem to copy constant detection. QBF (called QSAT in [10]) is a well-known PSPACE-complete problem.

Quantified Boolean Formulas. Let us first recall QBF. A QBF instance is a quantified Boolean formula,

$$\phi \equiv Q_n x_n : \cdots \forall x_2 : \exists x_1 : c_1 \wedge \ldots \wedge c_k \,,$$

where Q_n is the quantifier $\exists$ if n is odd and $\forall$ if n is even, i.e. quantifiers are strictly alternating.

[2] If the program has recursive procedures, the definition of runs is no longer inductive. Then the clauses are meant to specify the smallest sets obeying the given equations, which exist by the well-known Knaster-Tarski fixpoint theorem. However, only non-recursive programs occur in this paper.

As in SAT, each clause c_i is a disjunction of *literals*, where a literal l is either a variable from $X = \{x_1, \ldots, x_n\}$ or a negated variable from $\bar{X} = \{\neg x_1, \ldots, \neg x_n\}$. The set of indices of clauses made true by literal l is $\mathsf{Cl}(l) \overset{\text{def}}{=} \{i \in \{1, \ldots, k\} \mid c_i \text{ contains } l\}$. For later reference the following names are introduced for the sub-formulas of ϕ:

$$\phi_0 \equiv c_1 \wedge \ldots \wedge c_k \qquad \text{and} \qquad \phi_i \equiv Q_i x_i : \phi_{i-1} \quad \text{for } 1 \leq i \leq n\,,$$

where again Q_i is $\exists$ if i is odd and $\forall$ if i is even. Clearly, ϕ is just ϕ_n.

Formula ϕ_i is assigned a truth value with respect to a *truth assignment* $T \in \mathsf{TA}_i \overset{\text{def}}{=} \{T \mid T : \{x_{i+1}, \ldots, x_n\} \to \mathbb{B}\}$. We write $T[x \mapsto b]$ for the truth assignment that maps x to $b \in \mathbb{B}$ and behaves otherwise like T. We use this notation only if x is not already in the domain of T. For a truth assignment T we denote by $\mathsf{Cl}(T)$ the set of indices of clauses that are made true by T:

$$\mathsf{Cl}(T) \overset{\text{def}}{=} \bigcup_{x:\, T(x)=\mathsf{tt}} \mathsf{Cl}(x) \cup \bigcup_{x:\, T(x)=\mathsf{ff}} \mathsf{Cl}(\neg x)\,.$$

Note that $\mathsf{Cl}(T[x \mapsto \mathsf{tt}]) = \mathsf{Cl}(T) \cup \mathsf{Cl}(x)$ and $\mathsf{Cl}(T[x \mapsto \mathsf{ff}]) = \mathsf{Cl}(T) \cup \mathsf{Cl}(\neg x)$ (recall that x is not in the domain of T). Note also that TA_n contains only the trivial truth assignment $\emptyset$ for which $\mathsf{Cl}(\emptyset) = \emptyset$.

Using this notation, the truth value of a formula with respect to a truth assignment can be defined as follows:

$$T \models \phi_0 \ \text{ iff } \ \mathsf{Cl}(T) = \{1, \ldots, k\}$$
$$T \models \phi_i \ \text{ iff } \ \begin{cases} T[x_i \mapsto \mathsf{tt}] \models \phi_{i-1} \text{ or } T[x_i \mapsto \mathsf{ff}] \models \phi_{i-1}\,, & \text{if } i \text{ is odd } (Q_i = \exists) \\ T[x_i \mapsto \mathsf{tt}] \models \phi_{i-1} \text{ and } T[x_i \mapsto \mathsf{ff}] \models \phi_{i-1}\,, & \text{if } i \text{ is even } (Q_i = \forall) \end{cases}$$

The Reduction. From a QBF instance as above, we construct a program, in which we again use $k+1$ variables $z_0, z_1, \ldots, z_k$ in a similar way as in Sect. 3. Let the programs π_l be defined as in that section.

Let $\mathsf{Proc} = \{Main, P_0, P_1, \ldots, P_n\}$ be the set of procedures. The associated statements are defined as follows:

$$\pi_{Main} \overset{\text{def}}{=} z_0 := 1 \,;\, z_1 := 0 \,;\, \ldots \,;\, z_k := 0 \,;\, P_n \,;\, (z_0 := 0 \sqcap \mathbf{skip}) \,;\, \mathbf{write}(z_0)$$
$$\pi_{P_0} \overset{\text{def}}{=} z_1 := 0 \,;\, \ldots \,;\, z_k := 0 \,;\, z_0 := z_k$$
$$\pi_{P_i} \overset{\text{def}}{=} \begin{cases} (\pi^*_{x_i} \parallel P_{i-1}) \sqcap (\pi^*_{\neg x_i} \parallel P_{i-1})\,, & \text{if } i \text{ is odd} \\ (\pi^*_{x_i} \parallel P_{i-1}) \,;\, (\pi^*_{\neg x_i} \parallel P_{i-1})\,, & \text{if } i \text{ is even} \end{cases} \quad \text{for } 1 \leq i \leq n\,.$$

Clearly, this program can be constructed from the QBF instance in polynomial time or logarithmic space. Note that the introduction of procedures is essential for this to be the case. While we could easily construct an equivalent program without procedures by inlining the procedures, i.e. by successively replacing each call to procedure P_j by its body, for $j = 0, \ldots, n$, the size of the resulting program would in general be exponential in n, as each procedure P_j is called twice in P_{j+1}.

Therefore, we need the procedures to write this program succinctly and to obtain a logspace-reduction.

We show in the following, that the variable z_0 is not a constant at the write-statement in procedure *Main* if and only if the QBF instance is true. This establishes the PSPACE-hardness claim.[3]

Observe again that z_0 can hold only the values 0 and 1 at the write-statement because all variables are initialized by these values and the other assignments only copy them. Clearly, due to the non-deterministic choice just before the write-statement, it can hold 0. Thus, z_0 is a constant at the write-statement iff it cannot hold 1 there. Hence we can rephrase our proof goal as follows:

$$\begin{array}{l}z_0 \text{ can hold the value 1 at the write-statement in } \pi_{Main}\\ \text{if and only if } \phi \text{ is true.}\end{array} \qquad \text{(PG)}$$

In the remainder of this section we separately prove the 'if' and the 'only if' direction.

The "If" Direction. For the 'if' claim, we show that procedure P_n has a run of a special form called a copy chain, if ϕ is true.

Definition 3. *A* (total) segment *is a sequence of assignment statements of the form* $\langle z_1 := 0, \ldots, z_k := 0, (z_1 := z_0)^{n_1}, \ldots, (z_k := z_{k-1})^{n_k}, z_0 := z_k \rangle$, *where* $n_i \geq 1$ *for* $i = 1, \ldots, n$. *A* (total) copy chain *is a concatenation of segments.*

Every segment copies the initial value of z_0 back to z_0 via the sub-chain of assignments $z_1 := z_0, z_2 := z_1, \ldots, z_k := z_{k-1}, z_0 := z_k$, where each $z_i := z_{i-1}$ is the last assignment in the block $(z_i := z_{i-1})^{n_i}$. Note that the other statements in a segment do not kill this value; in particular the assignments $\langle z_1 := 0, \ldots, z_k := 0\rangle$ do not affect z_0. By induction on the number of segments, a total copy chain copies the initial value of z_0 back to z_0 too. Thus, if P_n has a run that is a total copy chain, then z_0 can, at the write-statement in π_{Main}, hold the value 1 by which it was initialized. As a consequence the following lemma implies the 'if'-direction of (PG).

Lemma 4. *If ϕ is true, then P_n has a run that is a total copy chain.*

In order to enable an inductive proof of this lemma we consider *partial copy chains* in which some of the blocks $(z_i := z_{i-1})^{n_i}$ may be missing (i.e. n_i may be zero).

Definition 5. *A* partial segment *is a sequence of assignment statments of the form* $s = \langle z_1 := 0, \ldots, z_k := 0, (z_1 := z_0)^{n_1}, \ldots, (z_k := z_{k-1})^{n_k}, z_0 := z_k \rangle$, *where now* $n_i \geq 0$ *for* $i = 1, \ldots, n$. *For* $H \subseteq \{1, \ldots, k\}$ *we say that s is a* partial segment with holes in H *if* $H \supseteq \{i \mid n_i = 0\}$. *A* partial copy chain with holes in H *is a concatenation of partial segments with holes in* H.

[3] Recall that PSPACE coincides with co-PSPACE because PSPACE is closed under complement.

Intuitively, the holes in a partial copy chain may be filled by programs running in parallel to form a total copy chain. Note that a partial copy chain with holes in $H = \emptyset$ is a total copy chain.

Lemma 6. *For all $i = 0, \ldots, n$ and all truth assignments $T \in \mathsf{TA}_i$ the following holds: if $T \models \phi_i$ then P_i has a partial copy chain with holes in $\mathsf{Cl}(T)$.*

Note that Lemma 6 indeed implies Lemma 4: ϕ is true iff the (unique) truth assignment $T \in \mathsf{TA}_n$, viz. $T = \emptyset$, satisfies ϕ_n. By Lemma 6, P_i has then a partial copy chain with holes in $\mathsf{Cl}(\emptyset) = \emptyset$, i.e. a total copy chain.

We show Lemma 6 by induction on i.

Base case ($i = 0$). Suppose given $T \in \mathsf{TA}_0$ with $T \models \phi_0$, i.e. $\mathsf{Cl}(T) = \{1, \ldots, k\}$. By definition, P_0 has the run $\langle z_1 := 0, \ldots, z_k := 0, z_0 := z_k \rangle$, which may be written as $\langle z_1 := 0, \ldots, z_k := 0, (z_1 := z_0)^0, \ldots, (z_k := z_{k-1})^0, z_0 := z_k \rangle$, i.e. it is a partial copy chain with holes in $\{1, \ldots, k\} = \mathsf{Cl}(T)$.

Induction step ($i \to i+1$). Assume that for a given i, $0 \le i \le k-1$, the claim of Lemma 6 holds for all $T \in \mathsf{TA}_i$ (induction hypothesis). Suppose given $T \in \mathsf{TA}_{i+1}$ with $T \models \phi_{i+1}$.

If $i+1$ is even, we have, by definition of ϕ_{i+1}, $T \models \forall x_i : \phi_i$, i.e. $T[x_{i+1} \mapsto \mathsf{tt}] \models \phi_i$ and $T[x_{i+1} \mapsto \mathsf{ff}] \models \phi_i$. By the induction hypothesis, there are thus two partial copy chains r_{tt} and r_{ff} with holes in $\mathsf{Cl}(T[x_{i+1} \mapsto \mathsf{tt}]) = \mathsf{Cl}(T) \cup \mathsf{Cl}(x_{i+1})$ and $\mathsf{Cl}(T[x_{i+1} \mapsto \mathsf{ff}]) = \mathsf{Cl}(T) \cup \mathsf{Cl}(\neg x_{i+1})$, respectively.

By interleaving each segment of r_{tt} with a single iteration of $\pi^*_{x_{i+1}}$ appropriately we can fill the holes from $\mathsf{Cl}(x_{i+1})$; this gives us a run r_1 of $\pi^*_{x_{i+1}} \parallel P_i$ that is a partial copy chain with holes in $\mathsf{Cl}(T)$. Similarly, we can fill the holes from $\mathsf{Cl}(\neg x_{i+1})$ in r_{ff} by interleaving each segment with an iteration from $\pi_{\neg x_{i+1}}$; this gives us a run r_2 of $\pi^*_{\neg x_{i+1}} \parallel P_i$ that is a partial copy chain with holes in $\mathsf{Cl}(T)$ too. By concatenating r_1 and r_2 we get a run of P_{i+1} that is a partial copy chain with holes in $\mathsf{Cl}(T)$.

The argumentation for the case that $i+1$ is odd is similar.

The 'Only If' Direction. As the constant 1 appears only in the initialization to z_0, z_0 can hold the value 1 finally in π_{Main} only if P_n has a run that copies z_0 (perhaps via other variables) back to z_0. We call such a run a *copying run.* Thus, the 'only if' direction of (PG) follows from the following lemma.

Lemma 7. *If P_n has a copying run then ϕ is true.*

Note that, while we could restrict attention to runs of a special form in the 'if'-proof, viz. total and partial copy *chains*, we have to consider arbitrary runs here, as any of them may copy z_0's initial value back to z_0.

In order to enable an inductive proof, we will be concerned with runs that are not (necessarily) yet copying runs but may become so if assignments from a set A are added at appropriate places. Each assignment from A may be added zero, one or many times. The assignment sets A considered are induced by truth assignments T: $A = \mathsf{Asg}(T) \stackrel{\text{def}}{=} \{z_i := z_{i-1} \mid i \in \mathsf{Cl}(T)\}$. We call such a run a *potentially copying run with holes in* $\mathsf{Asg}(T)$.

Lemma 8. *For all $i = 0, \ldots, n$ and for all $T \in \mathsf{TA}_i$ the following is valid: If there is a potentially copying run of P_i with holes in $\mathsf{Asg}(T)$ then $T \models \phi_i$.*

Note that the case $i = n$ establishes Lemma 7: For the empty truth assignment $\emptyset \in \mathsf{TA}_n$, we have $\mathsf{Asg}(\emptyset) = \emptyset$ and a potentially copying run with holes in $\emptyset$ is just a copying run. Moreover, $\emptyset \models \phi_n$ iff ϕ is true.

We show Lemma 8 by induction on i.

Base case ($i = 0$). Suppose given $T \in \mathsf{TA}_0$. The only run of P_0 is

$$r = \langle z_1 := 0, \ldots, z_k := 0, z_0 := z_k \rangle .$$

If r is a potentially copying run with holes in $\mathsf{Asg}(T)$, assignments from $\mathsf{Asg}(T)$ can be added to r in such a way that the initial value of z_0 influences its final value. As we have only assignments of the form $z_i := z_{i-1}$ available, this can only happen via a sub-chain of assignments of the form $z_1 := z_0, z_2 := z_1, \ldots, z_k := z_{k-1}$, where each assignment $z_i := z_{i-1}$ has to take place after $z_i := 0$ and $z_k := z_{k-1}$ must happen before the final $z_0 := z_k$. Therefore, all assignment $z_1 := z_0, \ldots, z_k := z_{k-1}$ are needed. This means that $\mathsf{Asg}(T)$ must contain all of them, i.e. $\mathsf{Cl}(T)$ must be $\{1, \ldots, k\}$. But then $T \models \phi_0$.

Induction step ($i \to i+1$). Suppose given i, $0 \le i \le k-1$, and $T \in \mathsf{TA}_{i+1}$. Assume that there is a potentially copying run r of P_{i+1} with holes in $\mathsf{Asg}(T)$.

If $i+1$ is odd, r is either a run of $\pi^*_{x_{i+1}} \parallel P_i$ or of $\pi^*_{\neg x_{i+1}} \parallel P_i$. We discuss the case $\pi^*_{x_{i+1}} \parallel P_i$ in detail; the case $\pi^*_{\neg x_{i+1}} \parallel P_i$ is analogous. So let r be an interleaving of a run s of $\pi^*_{x_{i+1}}$ and t of P_i. By definition of $\pi_{x_{i+1}}$, s consists only of assignments from $\mathsf{Asg}(x_{i+1}) \stackrel{\text{def}}{=} \{z_j := z_{j-1} \mid j \in \mathsf{Cl}(x_{i+1})\}$. As r can be interleaved with the assignments in $\mathsf{Asg}(T)$ to form a copying run, t can be interleaved with assignments from $\mathsf{Asg}(T) \cup \mathsf{Asg}(x_{i+1})$ to form a copying run. Therefore, t is a potentially copying run with holes in $\mathsf{Asg}(T) \cup \mathsf{Asg}(x_{i+1}) = \mathsf{Asg}(T[x_{i+1} \mapsto \mathsf{tt}])$. By the induction hypothesis thus $T[x_{i+1} \mapsto \mathsf{tt}] \models \phi_i$. Consequently, $T \models \exists x_{i+1} : \phi_i$, i.e. $T \models \phi_{i+1}$.

If $i+1$ is even, there are runs s and t of $\pi^*_{x_{i+1}} \parallel P_i$ and $\pi^*_{\neg x_{i+1}} \parallel P_i$ respectively, such that $r = s \cdot t$. It suffices to show that s and t are potentially copying runs with holes in $\mathsf{Asg}(T)$. An argumentation like in the case '$i+1$ odd' then yields that $T[x_{i+1} \mapsto \mathsf{tt}] \models \phi_i$ and $T[x_{i+1} \mapsto \mathsf{ff}] \models \phi_i$ and thus $T \models \forall x_{i+1} : \phi_i$, i.e. $T \models \phi_{i+1}$.

As $r = s \cdot t$ is a potentially copying run with holes in $\mathsf{Asg}(T)$ it may be interleaved with assignments from $\mathsf{Asg}(T)$ to form a copying run r'. Clearly, we can interleave its two parts s and t separately by assignments from $\mathsf{Asg}(T)$ to sequences s' and t' such that $r' = s' \cdot t'$. It is, however, not obvious that s' and t' really copy from z_0 to z_0 – if they do so, we are done because then s and t

are potentially copying runs with holes in $\mathsf{Asg}(T)$. Of course, there must be a variable z_j such that the value of z_0 is copied by s' to z_j and the value of z_j is copied by t' to z_0; otherwise z_0 cannot be copied to z_0 by r'. But, at first glance, z_j may be different from z_0. It follows from the below lemma, that z_j indeed must be z_0, which completes the proof of Lemma 8.

Lemma 9. *Let r be some interleaving of a run of P_i, $i = 0, \ldots, n$, with assignments of the form $z_l := z_{l-1}$, $l = 1, \ldots, k$. Then r copies none of the variables $z_1, \ldots, z_k$ to some variable.*

This last lemma is proved by induction on i. The interesting argument is in the base case; the induction step is almost trivial.

Base case. Let $i = 0$ and assume given a variable z_j, $j \in \{1, \ldots, k\}$. Then r is an interleaving of $\langle z_1 := 0, \ldots, z_k := 0, z_0 := z_k \rangle$ with assignments of the form $z_l := z_{l-1}$. Assignments of this form can copy only to variables with a higher index. Thus, just before the assignment $z_j := 0$ at most the variables $z_j, z_{j+1}, \ldots, z_k$ can contain the value copied from z_j. The contents of z_j is overwritten by the assignment $z_j := 0$. So immediately after $z_j := 0$ at most $z_{j+1}, \ldots, z_k$ can contain the value copied from z_j. This also holds just before the assignment z_{j+1} which overwrites z_{j+1}; and so on. Just after $z_k := 0$, no variable can still contain the value copied from z_j.

Induction step. Let $i > 0$ and assume that the claim is valid for $i - 1$. Any run of P_i either starts with (if i is even) or is (if i is odd) an interleaving of a run of P_{i-1} with assignments of the described form. Therefore, r starts with or is an interleaving of a run of P_{i-1} with such assignments. The property follows thus immediately from the induction hypothesis.

5 Conclusion

In this paper we have presented two complexity results with detailed proofs. They indicate that the accounts of [7, 6, 2, 3, 13] on efficient and complete data-flow analysis of parallel programs cannot be generalized significantly beyond gen/kill problems.

The reductions in this paper apply without change also to the *may-constant* detection problem in parallel programs. In the may-constant problem [9] we ask whether a given variable x can hold a given value k at a certain program point p or not, i.e. whether there is a run from the start of the program to p after which x holds k. In the NP-hardness proof in Sect. 3 we showed that z_k may hold the value 1 at the write-statement iff the given SAT instance is satisfiable and, similarly, in Sect 4 that z_0 may hold 1 at the write-statement iff the given QBF instance is true. This proves that the may constant problem is NP-complete for loop-free parallel programs and PSPACE-hard for programs with procedures and loops. Also the complexity of another data-flow problem, that of detecting faint variables [4] which is related to program slicing [16, 15], can be attacked with essentially the same reductions.

For the interprocedural parallel problem the current paper only establishes a lower bound, viz. PSPACE-hardness. It is left for future work to study the precise complexity of this problem. Another interesting question is the complexity of the general intraprocedural problem for parallel programs where we have loops but no procedures.

Acknowledgments. I thank Helmut Seidl for a number of discussions that stimulated the research reported here and Jens Knoop, Oliver Rüthing, and Bernhard Steffen for sharing their knowledge of data-flow analysis with me. I am also grateful to the anonymous referees for their valuable comments.

References

1. S. A. Cook. The complexity of theorem-proving procedures. In *ACM STOC'71*, pages 151–158, 1971.
2. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FOSSACS '99*, LNCS 1578, pages 14–30. Springer, 1999.
3. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *ACM POPL'2000*, pages 1–11, 2000.
4. R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *GI 11. Jahrestagung*, Informatik Fachberichte 50, pages 1–10. Springer, 1981.
5. G. A. Kildall. A unified approach to global program optimization. In *ACM POPL'73*, pages 194–206, 1973.
6. J. Knoop. Parallel constant propagation. In *Euro-Par'98*, LNCS 1470, pages 445–455. Springer, 1998.
7. 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.
8. S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.
9. R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analysis. In *ACM POPL'2000*, pages 67–81, 2000.
10. C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
11. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. Technical Report RC 21493, IBM T. J. Watson Research Center, 1999. To appear in TOPLAS.
12. J. R. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *ACM POPL'77*, pages 104–118, 1977.
13. H. Seidl and B. Steffen. Constraint-based interprocedural analysis of parallel programs. In *ESOP'2000*, LNCS 1782, pages 351–365. Springer, 2000.
14. R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.
15. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–181, 1995.
16. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.