The Complexity of Copy Constant Detection in Parallel Programs

Markus Müller-Olm

Universität Dortmund, FB Informatik, LS 5,
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 bit-vector and gen/kill problems can be solved efficiently and completely for parallel programs with a shared state [4, 3, 9]. The results of this paper show that it is unlikely that a generalization to significantly larger classes of data-flow problems is possible.
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. More specifically, we show that already the intra-procedural problem for acyclic parallel programs is co-NP-complete and that the inter-procedural 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 [4] show that *bit-vector analyses*, which comprise, e.g., live/dead variable analysis, available expression analysis, and reaching definition analysis [5], can efficiently be performed on such programs. They restrict attention to the *intra-procedural* problem, in which each procedure body is analyzed separately with worst case assumption on called procedures. Seidl and Steffen [9] generalize this to the inter-procedural case in which the interplay between procedures is taken into account. They also consider a slightly more extensive class of data-flow problems called *gen/kill problems* which 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. The results of [9] also subsume the ones of [4], in which it is shown that a simple variant of constant detection, *strong simple constants*, is tractable in the intra-procedural case for parallel programs.

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 non-parallel 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 [2] – completely and efficiently be solved by a fixpoint computation. We show in this paper that already for acyclic parallel programs without procedures the problem is co-NP-complete and becomes even PSPACE-hard if one allows loops and procedures. This renders the possibility of complete and efficient data-flow algorithms for parallel programs for more extensive classes of analysis unlikely, as it is generally believed that the inclusions P $\subseteq$ co-NP $\subseteq$ PSPACE are proper.

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 for example for the question whether a condition in a program may be satisfied or not. In order to come to grasp with undecidability, it is, therefore, common in data-flow analysis to abstract from the conditions in the programs. Thus conditional branching is interpreted as non-deterministic branching. We adopt this point of view in this paper and restrict attention to non-deterministically branching programs. Of course, an analysis based on this abstraction considers more executions than actually possible at run-time. One is usually careful to take this into account when exploiting the results of the data-flow analysis.

A standard transformation in optimizing compilers is *constant propagation*. It aims at replacing expressions in the program that evaluate to one and the same value in every possible run of the program by a constant denoting this very value. This transformation is profitable as it decreases both code size and execution time. The associated data-flow analysis problem is to detect constant expressions in the program ('must-constants' in the sense of [6]).

It has been shown by Reif and Lewis [8] by a reduction of Hilbert's tenth problem that the general constant detection problem is undecidable, even if branching is interpreted non-deterministically. However, if one restricts the kind of expressions allowed on the right hand side of assignments appropriately, the problem becomes decidable (see [6] for results on the complexity of constant detection for various classes of sequential programs.) A problem that is particularly simple for sequential programs are so-called *copy constants*. In this problem we restrict attention to programs were assignments 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 practice assignments of a different form are treated by approximating or worst-case assumptions.

3 Acyclic Parallel Programs

Let us first of all set the stage for the intra-procedural copy constant detection problem. We consider *acyclic programs with (fork/join) parallelism* given by the following grammar,

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

where x ranges over some set of variables and c over some set of basic constants. Note that this language has only constant and copying assignments. The specific nature of constants and the domain in which they are interpreted is immaterial; we only need that 0 and 1 are two constants denoting two different values, which we also call 0 and 1. The operator $\sqcap$ represents non-deterministic choice; the interpretation of the remaining operator should be clear. Strictly speaking, write-statements are unnecessary for the current paper. But they are convenient for pointing clearly to the program point where we are interested in constancy of a variable.

Parallelism is understood in an interleaving fashion; assignments and write-statements are assumed to be atomic. A run of a program is a 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 "$\cdot$". The concatenation operator is lifted to sets of sequences as follows: if S, T are two sets of sequences then $S \cdot T = \{s \cdot t \mid s \in S, t \in T\}$. Let $r = \langle e_1, \ldots, e_n\rangle$ be a sequence and $I = \{i_1, \ldots, i_k\}$ a subset of positions in r such that $i_1 < i_2 < \cdots < i_k$. Then $r|I$ is the word $\langle e_{i_1}, \ldots, e_{i_k}\rangle$ and we write $|r|$ for the length of r, viz. n. The *interleaving* of S and T is

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

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

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

4 The co-NP Result

Theorem 1. *The problem of detecting copy constants in an acyclic parallel program is co-NP-complete.*

Certainly, the problem lies in co-NP: if a variable x is *not* constant at a certain place in the program we can guess two paths that witness two different values. As the program is acyclic, the length of these paths (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 3-SAT, a well-known NP-complete problem [7], to the negation of a copy constant detection problem. The same reduction also applies to the general SAT problem.

Let us first recall the 3-SAT problem. An instance of 3-SAT is a conjunction $c_1 \wedge \ldots \wedge c_k$ of *clauses* $c_1, \ldots, c_k$. Each clause is a disjunction of three *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 $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$, satisfies $c_1 \wedge \ldots \wedge c_k$. The 3-SAT problem asks us to decide for each instance $c_1 \wedge \ldots \wedge c_k$ whether there is a satisfying truth assignment or not.

Now suppose given a 3-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 3-SAT instance we construct an acyclic parallel program. In the program we use $k+1$ variables $z_0, z_1, \ldots, z_k$. Intuitively, z_i is, for $1 \le i \le k$, 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

$$\pi_l^0 \stackrel{\text{def}}{=} \textbf{skip}$$

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

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

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

Clearly, π can be constructed from the given 3-SAT instance $c_1 \wedge \ldots \wedge c_k$ in polynomial time (or even logarithmic space). We show that the variable z_k at the write-statement is not 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 $T(x_i) = \mathsf{tt}$ 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, in which the value 1 with which z_0 is initialized is copied 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.

5 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 is associated with a statement π_P constructed according to the grammar

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

where Q ranges over Proc. A statement of the form Q represents a call to procedure Q and π^* a loop that iterates π an indefinite number of times. Looping is indefinite due to the abstraction that branching is non-deterministic.

The definition of runs from the previous section can easily be extended to the enriched language by the two clauses:[1]

$$\begin{aligned} \mathsf{Runs}(\pi^*) &= \mathsf{Runs}(\pi)^* \\ \mathsf{Runs}(P) &= \mathsf{Runs}(\pi_P) \end{aligned}$$

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*.

[1] 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. The programs in this paper, however, will not contain recursive procedures.

6 The PSPACE-Hardness Result

Theorem 2. *The problem of detecting copy constants in procedural parallel programs with parallelism and loops is PSPACE-hard.*

In the remainder of this section we prove this theorem by means of a reduction of the QBF (quantified Boolean formulas) problem to copy constant detection. QBF is a well-known PSPACE-complete problem [7].

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.

As in 3-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 $\bar{X} = \{\neg x_1, \ldots, \neg x_n\}$. We denote by $\mathsf{Cl}(l)$ the set of indices of clauses made true by literal l:

$$\mathsf{Cl}(l) \stackrel{\text{def}}{=} \{i \in \{1, \ldots, k\} \mid c_i \text{ contains } l\} .$$

For later reference let us introduce names for the sub-formulas of ϕ:

$$\begin{aligned} \phi_0 &\equiv c_1 \wedge \ldots \wedge c_k \\ \phi_i &\equiv Q_i x_i : \phi_{i-1} \quad \text{for } 1 \le i \le n \end{aligned}$$

where 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* $T \in \mathsf{TA}_i \stackrel{\text{def}}{=} \{T \mid T : \{x_{i+1}, \ldots, x_n\} \to \mathbb{B}\}$. 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:

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

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 Construction. 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:

$$\pi_{Main} \stackrel{\text{def}}{=} z_0 := 1 \,;\, z_1 := 0 \,;\, \ldots \,;\, z_k := 0 \,;\, P_n \,;\, (z_0 := 0 \sqcap \mathbf{skip}) \,;\, \mathbf{write}(z_0)$$

$$\pi_{P_0} \stackrel{\text{def}}{=} z_1 := 0 \,;\, \ldots \,;\, z_k := 0 \,;\, z_0 := z_k$$

$$\pi_{P_i} \stackrel{\text{def}}{=} \begin{cases} (\pi^*_{x_i} \parallel P_{i-1}) \sqcap (\pi^*_{\neg x_i} \parallel P_{i-1})\,, & \text{if } i \text{ is odd} \\ (\pi^*_{x_i} \parallel P_{i-1}) \,;\, (\pi^*_{\neg x_i} \parallel P_{i-1})\,, & \text{if } i \text{ is even} \end{cases} \quad \text{for } 1 \leq i \leq n\,.$$

Clearly, this program can be constructed from the QBF instance in polynomial time or logarithmic space. We show in the following, that the assignment 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.[2]

Observe again that 0 and 1 are the only values z_0 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, it can hold 0. Thus, z_0 is a constant at the write-statement iff it cannot hold 1 there. Thus, we can rephrase our proof goal as follows: z_0 can hold the value 1 at the write-statement if and only if ϕ is true.

The "If" Direction. For the 'if' claim, we are going to show that, if ϕ is true, P_n has a run consisting of a concatenation of *segments* 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 $i = 1, \ldots, n$. We call such a run a *full dependency chain*. 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 $(z_i := z_{i-1})^{n_i}$. After a full dependency chain, z_0 holds, thus, the value 1 assigned to it by the initialization $z_0 := 1$ in π_{Main}.

In order to enable an inductive proof we also consider *partial dependency chains*. These are dependency chains in which some of the blocks $(z_i := z_{i-1})^{n_i}$ may be missing (i.e. n_i may be zero). Formally, we call a sequence of assignments 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 now n_i may be zero, a *partial segment*. For $H \subseteq \{1, \ldots, k\}$ we say that s is a *segment with holes in H* if $H \supseteq \{i \mid n_i = 0\}$. Note that H may contain more indices than actually missing assignments. We call a sequence of assignments a *partial dependency chain with holes in H* if it is a concatenation of one or more

2 Recall that PSPACE coincides with co-PSPACE because PSPACE is closed under complement.

segments with holes in H. Intuitively, the holes in a partial dependency chain may be filled by programs running in parallel to form a full dependency chain. It is easy to see that a partial dependency chain with holes in $H = \emptyset$ is a full dependency chain.

Now, we are going to show, by induction on i, that for all $i = 0, \ldots, n$ the following is true for all truth assignments $T \in \mathsf{TA}_i$:

$$\text{if } T \models \phi_i \text{ then } P_i \text{ has a partial dependency chain with holes in } \mathsf{Cl}(T)\,. \qquad (1)$$

Note that this implies indeed that P_n has a full dependency chain if ϕ is satisfiable: ϕ is satisfiable iff the (unique) truth assignment $T \in \mathsf{TA}_n$, viz. $T = \emptyset$, satisfies ϕ_n. By (1), P_i has then a partial dependency path with holes in $\mathsf{Cl}(\emptyset) = \emptyset$, i.e. a full dependency path.

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 dependency chain with holes in $\{1, \ldots, k\} = \mathsf{Cl}(T)$.

Induction step ($i \to i + 1$). Assume that for a given i, $0 \le i \le k-1$, (1) 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 the 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 dependency 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 dependency 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 dependency 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 dependency chain with holes in $\mathsf{Cl}(T)$.

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

The ‘Only If’ Direction. Now, for the ‘only if’-claim. We are going to show that ϕ is true if P_n has a run that copies the initial value of z_0 back to z_0. It is easy to see that the latter is the case if and only if z_0 can hold a 1 finally. We call such a run a *full dependency path*. In the ‘if’ proof we constructed a certain computation of P_n. In this construction we could restrict attention to runs of a certain nice form (partial and full dependency *chains*) as we could construct even runs of this particular form. Here, however, we have to consider all possible runs, as any run may mediate a dependency between z_0’s initial and final value.

In order to enable an inductive proof, we will be concerned with runs that are not (necessarily) yet full dependency paths but can become so by adding assignments from a set A 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: $A = \mathsf{Asg}(T) = \{z_i := z_{i-1} \mid i \in \mathsf{Cl}(T)\}$. We call such a run a *partial dependency path with holes in* $\mathsf{Asg}(T)$.

We show, by induction on i, that for all $i = 0, \ldots, n$ and for all $T \in \mathsf{TA}_i$ the following is valid:

> If there is a partial dependency path of P_i with holes in $\mathsf{Asg}(T)$, then $T \models \phi_i$.

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 partial dependency path with holes in $\mathsf{Asg}(T)$ it must be possible to add assignments from $\mathsf{Asg}(T)$ to r such 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, the dependency can only be transferred via a 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 $T \models \phi_0$ holds.

Induction step. Suppose given i, $0 \leq i \leq k-1$, and $T \in \mathsf{TA}_{i+1}$. Assume that there is a partial dependency path r through P_{i+1} with holes in $\mathsf{Asg}(T)$.

If $i+1$ is odd, r is either a path 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. r is an interleaving of a path s of $\pi_{x_{i+1}}$ and t of P_i. By definition of $\pi_{x_{i+1}}$, s contains only assignments from $\mathsf{Asg}(x_{i+1}) := \{z_i := z_{i-1} \mid i \in \mathsf{Cl}(x_{i+1})\}$. As r can be interleaved with the assignments in $\mathsf{Asg}(T)$ to form a full dependency path, t can thus be interleaved with assignments from $\mathsf{Asg}(T) \cup \mathsf{Asg}(x_{i+1})$ to form a full dependency path. Therefore, t is a partial dependency path 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 paths 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 partial dependency paths 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 partial dependency path of $\pi^*_{x_{i+1}} \parallel P_i$ with holes in $\mathsf{Asg}(T)$ it may be enriched by assignments from $\mathsf{Asg}(T)$ to form a full dependency path r'. Clearly, we can enrich its two parts s and t separately by assignments from $\mathsf{Asg}(T)$ to paths s' and t' such that $r' = s' \cdot t'$. It is, however, not obvious that s' and t' really transfer dependencies from z_0 to z_0 (if they would do so, we would be done because then they would be partial dependency paths with holes in $\mathsf{Asg}(T)$.) Of course, there must be a variable z_j such that the initial value of z_0 is copied by s' to z_j and the value of z_j before t' is copied by t' to z_0; otherwise there 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.

Lemma 3. *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 none of the variables $z_1, \ldots, z_k$ is copied to some variable by r.*

Proof. The proof is by induction on i. The interesting argument is in the induction base; 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 an interleaving of a run of P_{i-1} with such assignments. The property follows thus immediately from the induction hypothesis. $\square$

7 Conclusion

In this paper we have presented two complexity results with detailed proofs. They indicate that the accounts of [4, 3, 9] cannot be generalized significantly.

The reductions in this paper apply without change also to the *may-constant* detection problem in parallel programs. In the may constant problem [6] we ask whether a given variable x can hold a certain value k at a certain program point or not. For the reduction in Sect. 3 we showed that z_k may hold the value 1 at the write-statement if the given 3-SAT instance is satisfiable and, similarly, in Sect 5 that z_0 may hold 1 at the write-statement if the given QBF instance is true. This proves that the may constant problem is NP-complete for acyclic parallel programs and PSPACE-hard for programs with procedures and loops. Also the complexity of another data-flow problem, that of detecting faint variables [1], can be attacked with essentially the same reductions.

Up to now, we have only a lower bound on the complexity of the interprocedural problem, viz. PSPACE-hardness. It is left for future work to establish an upper bound. Another interesting question is the complexity of the general intra-procedural problem for parallel programs where we have loops but no procedures. Here, we readily obtain a PSPACE upper bound but we currently know no lower bound. If, however, we assume atomic multiple assignment statements $x_1, \ldots, x_k := e_1, \ldots, e_k$, we can show PSPACE-hardness. This even holds if we allow only multiple assignments of the form $x_1, \ldots, x_k := 0, \ldots, 0$ (synchronized initialization/reset).

Acknowledgments. I thank Helmut Seidl for raising the question whether faint variables can be detected efficiently and completely in parallel programs and for refuting on the spot my naïve response. My interest in the problems studied in this paper resulted from these discussions. I am also grateful to Jens Knoop, Oliver Rüthing, and Bernhard Steffen who shared their knowledge on data-flow analysis with me and introduced me to the results about bit-vector analyses for parallel program.

References

1. S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data-flow analysis. *Acta Informatica*, 24:679–694, 1987.
2. G. A. Kildall. A unified approach to global program optimization. In *Conf. Rec. 1st ACM Symposium on Principles of Programming Languages POPL'73*, pages 194–206, 1973.
3. Jens Knoop. Parallel constant propagation. In D. Pritchard and J. Reeve, editors, *4th European Conference on Parallel Processing (Euro-Par)*, volume 1470 of *Lecture Notes in Computer Science*, pages 445–455. Springer-Verlag, 1998.
4. Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.
5. Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.
6. Markus Müller-Olm and Oliver Rüthing. On the complexity of constant propagation. In this volume, 2000.
7. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
8. John R. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Conf. Rec. 4th ACM Symposium on Principles of Programming Languages POPL'77*, pages 104–118, Los Angeles, CA, January 1977.
9. Helmut Seidl and Bernhard Steffen. Constraint-based inter-procedural analysis of parallel programs. In Gert Smolka, editor, *Programming Languages and Systems (ESOP'2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, 2000.